اصول طراحی کامپایلرها

دانشگاه آزاد اسلامی واحد شهر قدس

دانشکده فنی و مهندسی

گروه کامپیوتر

فهرست

فصل چهارم - تجزیه بالا به پایین

تحلیلگر ذهن

ایجاد الگوریتم تحلیل نحوی بر مبنای عملکرد ذهن

نتیجه تحلیل عملکرد ذهن

گرامرهای **LL(1)**

مجموعه های سرآغاز و پیرو

تبدیل گرامرها به فرم **LL(1)**

فاکتور گیری چپ

تبدیل قواعد خود بازگشتی چپ

حذف قوانین تهی

ایجاد جدول تجزیه بالا به پایین

تجزیه گرهای کاهینه بازگشتی

بهبود از خطا

مولد تحلیلگر نحوی

تمرین

فصل پنجم - تجزیه پایین به بالا

مقدمه

اصول تجزیه

طرح روشی برای ایجاد جدول تجزیه **LR(1)**

تولید الگوریتم

الگوریتم تولید گراف تجزیه **LR(1)**

ایجاد جدول تجزیه

مشکل گرامرهای **LR(1)**

گرامرهای **LALR(1)**

گرامرهای **SLR(1)**

گرامرهای مبهم

تمرین

فصل ششم - تحلیل مفهومی

تحلیل مفهومی

فصل هفتم - تولید کد میانی

تولید کد میانی

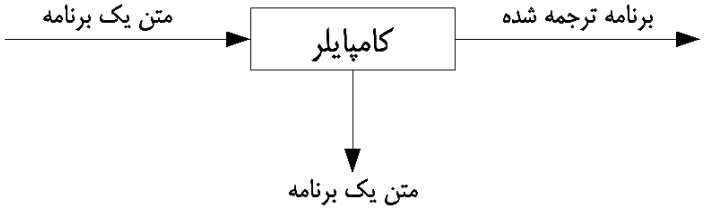
تولید کد اسمبلی

فصل اول

مبانی کامپایلر

۱-۱- تعریف کامپایلر

کامپایلر برنامه‌ای است که در ورودی خود، متن یک برنامه را که طبق قوانین و دستور زبان تدوین شده برای آن کامپایلر مشخص شده است، پذیرفته و در خروجی برنامه‌ای به زبان دوم ایجاد می‌نماید. این زبان دوم می‌تواند زبان ماشین یا برنامه به زبان دیگر باشد.

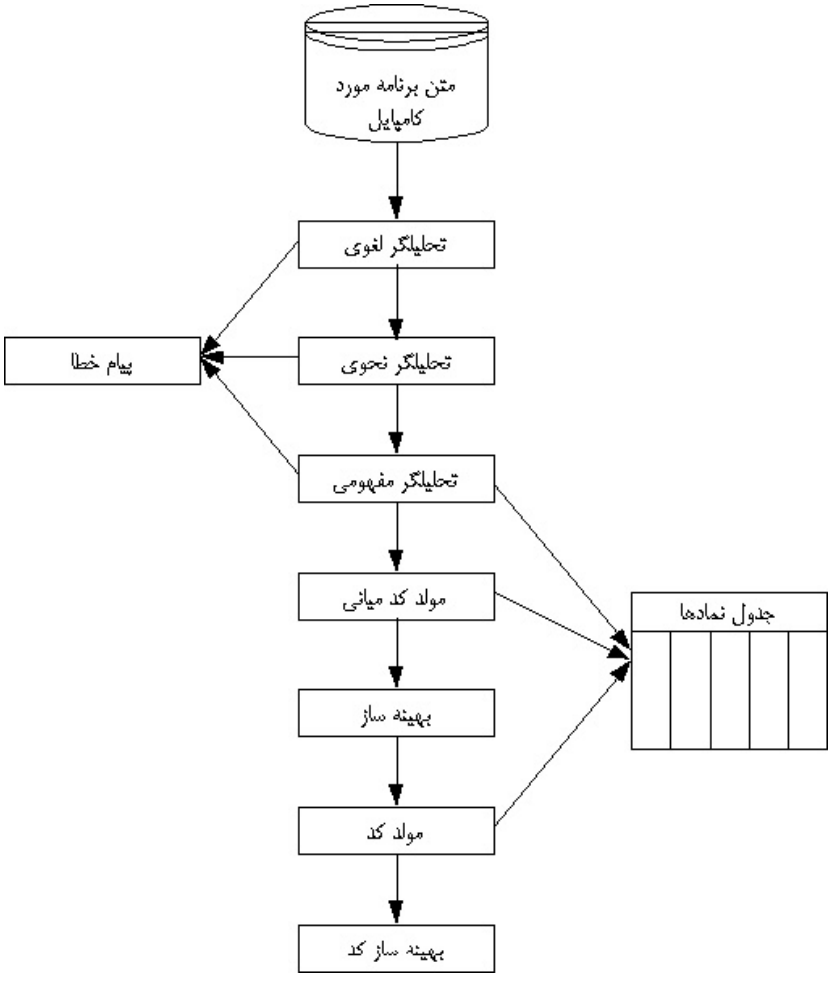


برنامه ترجمه شده در بعضی محیط‌ها مثل Unix معمولاً به زبان C می‌باشد. برای مثال کامپایلر راتفورد که نوع پیشرفته تر از فرترن می‌باشد در خروجی خود برنامه به زبان C را ایجاد می‌کند که توسط کامپایلر C تبدیل به کد ماشین می‌گردد.

بعضی از مترجم‌ها که در اصطلاح مفسر یا Interpreter نامیده می‌شوند همگام با ترجمه هر جمله اجرایی، آن را به اجرا در می‌آورند. البته این ترجمه همراه با اجرا، زمان اجرا را بسیار طولانی می‌کند اما امکاناتی در اختیار برنامه‌نویس قرار می‌دهد که کامپایلرهای عادی قادر به حصول آن نیستند. عمل کامپایل در طی مراحل مشخص که در بخش بعدی در مورد آن بحث خواهد شد، انجام می‌گردد. این مراحل می‌توانند به طور همزمان انجام شوند.

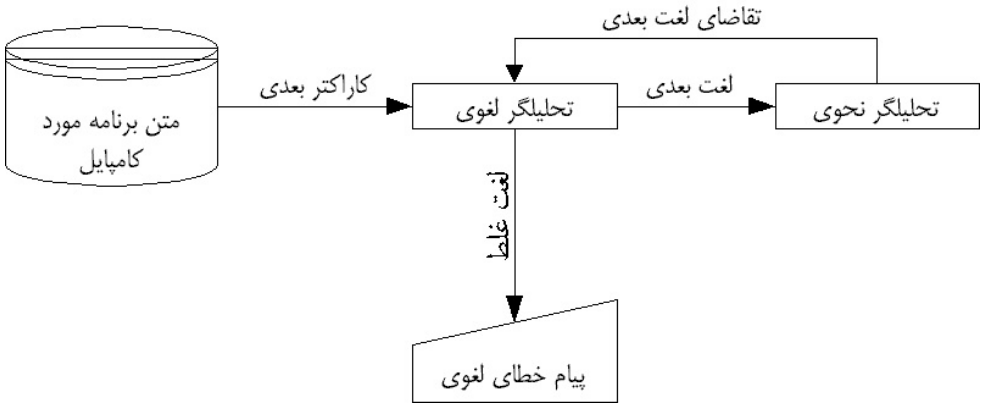
۱-۲- تعریف کامپایلر

همانطور که در شکل زیر مشخص شده است، کامپایلرها در حالت کلی از هفت بخش اصلی تشکیل می‌شوند. در این بخش به طور خلاصه بخش‌های متفاوت کامپایلر تشریح می‌شود.



الف- تحلیلگر لغوی

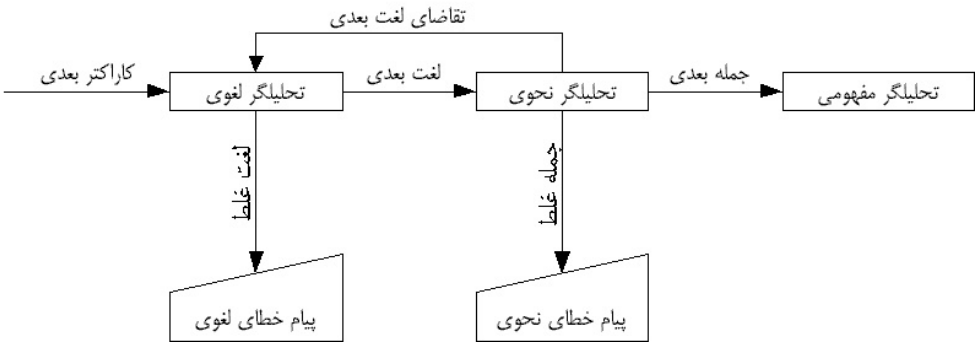
وظیفه تابع تحلیلگر لغوی (Lexical Analyser)، تشخیص لغات در متن برنامه مورد کامپایل است. این تابع توسط بخش دیگری از کامپایلر تحت عنوان تحلیلگر نحوی مورد فراخوانی قرار می‌گیرد. با هر بار فراخوانی، این تابع لغت بعدی را از متن برنامه تشخیص می‌دهد و اطلاعات لازم در مورد لغت را برای تحلیلگر نحوی ارسال می‌دارد. هر زبان برنامه‌سازی قوانین لغوی مربوط به خود را دارد. قوانین لغوی بیانگر فرم کلی انواع لغات در زبان برنامه‌سازی است. تحلیلگر لغوی در صورت وجود خطا در قالب بندی لغات استفاده شده در متن برنامه مورد کامپایل، اعلام خطای لغوی می‌نماید. تابع تحلیلگر لغوی در فصل دوم از این جزوه مورد بررسی قرار خواهد گرفت.



ب- تحلیلگر نحوی

وظیفه تحلیلگر نحوی(Syntax Analyser) تشخیص صحت فرم ظاهری برنامه‌ها از لحاظ دستورالعمل زبان برنامه‌سازی مربوطه است. تحلیلگر نحوی با فراخوانی تحلیلگر لغوی، لغات را از متن برنامه مورد کامپایل دریافت و صحت قرار گرفتن آنها در کنار یکدیگر بر اساس دستورالعمل زبان مربوطه را مورد آزمون و تحلیل نحوی قرار می‌دهد. در صورتی که از لحاظ دستورالعمل زبان، برنامه مورد کامپایل دارای خطا باشد، پیام خطا توسط تحلیلگر نحوی صادر می‌گردد.

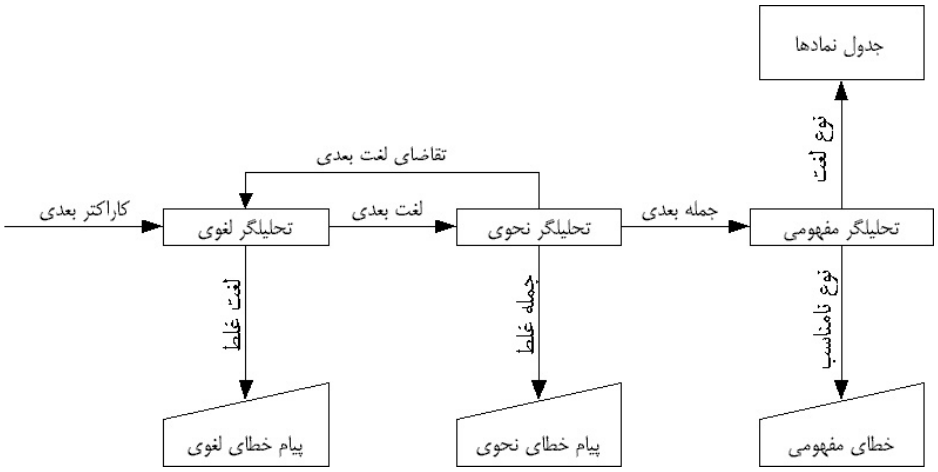
انواع روش‌های تحلیل نحوی و تشخیص صحت برنامه‌ها بر اساس دستورالعمل و گرامر زبانهای برنامه‌سازی در فصل‌های ۳ و ۴ و ۵ مورد بررسی قرار خواهد گرفت.



پ- تحلیلگر مفهومی

کار تحلیلگر مفهومی(Semantic Analyser) تعیین صحت مفهوم جملات است. ممکن است یک جمله از نظر نحوی صحیح ولی از لحاظ مفهومی دارای خطا باشد. برای مثال جمله "*مهرداد آتش را خورد*"، علیرغم اینکه در دستور زبان فارسی از نظر نحوی صحیح می‌باشد از نظر مفهومی نادرست است. در کامپایلرها به طور معمول عمل تحلیل مفهومی محدود به آزمون نوع(Type Checking) است. تحلیلگر مفهومی وابسته به نوع اسامی و متغیرها که در جدول نمادها(symbols table) مشخص شده، صحت استفاده آنها را

مورد آزمون قرار می‌دهد. برای مثال اگر متغیری از نوع رشته تعریف شده باشد، نمی‌توان آن را با متغیری از نوع صحیح یا اعشاری جمع یا مقایسه نمود. فصل ۷ در ارتباط با آزمون نوع است.

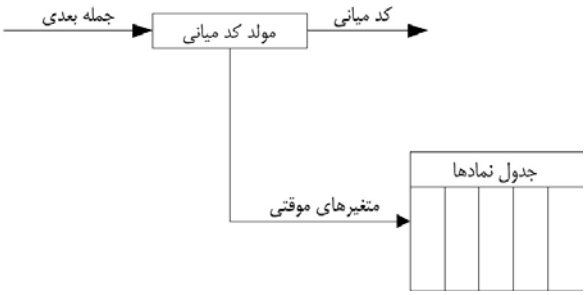


ت- مولد کد میانی

مولد کد میانی بخشی از کامپایلر است که در ورودی خود جملات تشخیص داده شده توسط تحلیلگر نحوی را پذیرفته و در خروجی کد واسطه یا میانی تولید می‌کند. کد میانی به سادگی قابل تبدیل و نزدیک به زبان ماشین است اما مستقل از ساختار و جزییات هر گونه ماشین خاص می‌باشد.

یکی از مزیت‌های ایجاد کد میانی افزایش قابلیت حمل کامپایلر بر روی سخت افزار با کد متفاوت و ساختار متفاوت است. به این ترتیب که برنامه کامپایلر را بتوان بر روی کامپیوترها با کد ماشین و اسمبلی متفاوت کامپایل نموده، با اجرای آن متن برنامه‌های مورد کامپایل را به کد میانی تبدیل کرد و سپس با نوشتن یک برنامه کوچک این کد واسطه را به کد ماشین مورد نظر تبدیل نمود. مزیت دیگر ایجاد کد میانی، فراهم نمودن شرایط خوب برای بهینه‌سازی کد برنامه‌هاست. کد میانی در فصل ۶ مطرح شده است.

ث- بهینه ساز کد میانی



هذف از بهینه‌سازی کد میانی می‌تواند تقلیل حجم و افزایش سرعت اجرایی کد ماشین حاصل از حاصل کار کامپایلر باشد. برای این منظور بهینه ساز، کد میانی حاصل از مولد کد میانی را مورد تحلیل قرار می‌دهد. به طور معمول تحلیل کد میانی با آزمون نمادی برنامه‌ها تحقق می‌یابد. به این ترتیب که مفهوم برنامه‌ها در زمان کامپایل، با تبدیل کد میانی به یک گراف به نام گراف جریان مورد بررسی و تحلیل واقع شده، سعی می‌کنند تا حجم کد حاصل را تقلیل داده، در صورت امکان کد زاید را مشخص و حذف نماید. بهینه‌سازی موضوع فصل ۸ این کتاب است.

ج- مولد کد

تبدیل کد میانی به کد زبان ماشین وظیفه این قسمت از کامپایلر است. این کد وابسته به سخت‌افزار بوده و می‌تواند براحتی به فایلهای قابل اجرا تبدیل گردد و اجرا شود.

چ- بهینه ساز کد

هدف از بهینه‌سازی کد تقلیل حجم اجرایی برنامه است. در این مرحله برای انجام عمل بهینه‌سازی و رسیدن به اهداف آن ساختار اسمبلی و کد ماشین و امکانات سخت افزاری آن در نظر گرفته می‌شود و از دستورالعمل‌هایی که سریعتر و با حجم کمتر به اجرا در می‌آیند، استفاده می‌شود. بهینه ساز کد در صورت امکان دستورالعمل‌های سریع و کم حجم را با دستورالعمل‌های به کار برده شده در خروجی مولد کد که همان کد اجرایی برنامه است جایگزین می‌کند.

البته مراحل فوق الذکر بعضاً ممکن است در ایجاد کامپایلر حذف شود. برای مثال قسمت بهینه‌سازی و یا تحلیل مفهومی ممکن است در یک کامپایلر وجود نداشته باشد. و نیز ممکن است کلیه مراحل در یک مرحله و به صورت همزمان اجرا شود. معمولاً عملیات کامپایل برنامه‌ها در دو و یا در اصطلاح در دو pass انجام می‌شود.

۱-۳- کامپایلر کامپایلرها

کامپایلر کامپایلر در واقع یک مولد کامپایلر می‌باشد و برنامه ای است که در ورودی خود قوانین لغوی و گرامری زبان را در ورودی گرفته و در خروجی خود یک کامپایلر ایجاد می‌کند.

در طی فصلهای بعدی خواهیم دید که چگونه می‌توان یک مولد تحلیلگر لغوی و تحلیلگر نحوی را به سادگی تولید نمود. از کامپایلر کامپایلرهای رایج یکی YACC را می‌توان نام برد. این مولد کامپایلر قواعد گرامری و کد مورد نظر برای تبدیل و در واقع کامپایل جملات متن برنامه کامپایلر را در ورودی خود دریافت می‌کند. خروجیYACC برنامه کامپایلر به زبان C است. عمل تحلیل لغوی برای این کامپایلر کامپایلر توسط یک مولد تحلیلگر لغوی lex انجام می‌شود.

۱-۴- کامپایلر برنامه‌ها با لغات فارسی

در زبان فارسی به علت اینکه عبارات از چپ به راست و جملات از راست به چپ نوشته می‌شوند نمی‌توان به سادگی با استفاده از روشهای جاری کامپایلر زبان برنامه‌سازی که لغات در آن به زبان فارسی هستند، ایجاد نمود. اگر دستورالعمل‌های زبان برنامه‌سازی با استفاده از لغات فارسی از چپ به راست نوشته شوند این مشکل بر طرف خواهد شد.

می‌توان برنامه‌های C یا پاسکال و هر زبان برنامه‌سازی دیگری را با استفاده از لغات فارسی و به زبان فارسی نوشت و با استفاده از یک تحلیلگر لغوی و جداول لغات کلیدی فارسی و مشابه آن در انگلیسی به زبان اصلی تبدیل نمود. پس از اشکال زدایی و تکمیل برنامه‌ها می‌توان بالعکس عمل نموده، متن انگلیسی را به فارسی تبدیل نمود.

پرسش و پاسخ

۱-۱ : تعریفی از کامپایلر ارائه دهید ؟

۱-۲: کار تابع تحلیلگر لغوی را شرح دهید ؟

۱-۳: کار تابع تحلیلگر نحوی را بیان کنید ؟

۱-۴: کار تابع تحلیلگر مفهومی را توضیح دهید ؟

۱-۵ : مولد کد میانی چیست ؟

۱-۶ : بهینه‌سازی کد میانی به چه منظور انجام می‌گیرد ؟

۱-۷ : هدف از بهینه‌سازی کد چه می‌باشد ؟

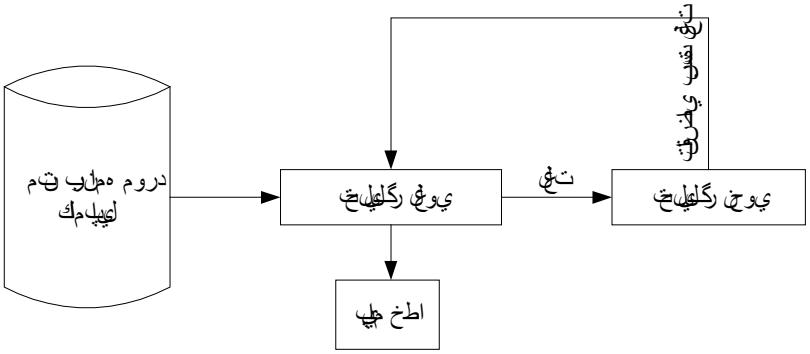
۱-۸ : کامپایلر کامپایلرها چیست ؟

فصل دوم

تحلیلگر لغوی

۲-۱- مقدمه

تحلیلگر لغوی تابعی است که در ورودی خود برنامه مورد کامپایل را به عنوان یک فایل متن باز گشوده و کاراکتر به کاراکتر می‌خواند و با رسیدن به یک کاراکتر جدا کننده، لغت را تشخیص و تفکیک می‌نماید. جدا کننده‌ها مثل blank و tab یا new line فقط به منظور جداسازی لغات به کار می‌روند. دسته دیگر از جدا کننده‌ها مثل semicolon یا + ارزش لغوی دارند. تحلیلگر لغوی در خروجی خود اطلاعات مربوط به لغت را در یک رکورد،ساختار یا کلاس به نام بسته لغت قرار می‌دهد. بسته لغت شامل اطلاعاتی در مورد لغت تشخیص داده شده توسط تحلیلگر لغوی مانند سطر یا ستون و نوع لغت را در اصطلاح token می‌گویند.



هر زبان برنامه سازی قوانین لغوی خاص خود را دارد. قوانین لغوی زبانها را می‌توان به صورت غیر مبهم و خلاصه در قالب نوعی خاص از عبارات به نام عبارت باقاعده بیان نمود. همچنین قوانین لغوی را می‌توان توسط نوعی خاص از ساختار گراف به نام ماشین‌های خودکار مطرح کرد. ماشینهای خودکار ابزاری برای تبدیل قوانین لغوی به کد برنامه هستند. چنانچه لغتی بنابر قوانین لغوی زبان برنامه‌سازی ایجاد نشده باشد، تحلیلگر لغوی اعلام خطا می‌نماید.

۲-۲- ساختار ورودی/خروجی

در داخل تحلیلگر لغوی، برنامه مورد کامپایل به عنوان یک فایل متنی خوانده می‌شود. این فایل در ابتدای برنامه کامپایلر و خارج از محیط تحلیلگر لغوی تعریف و گشوده می‌شود و به عنوان یک پارامتر به تابع تحلیلگر لغوی ارسال می‌گردد. در قطعه کد ارائه شده در زیر چگونگی فراخوانی تابع تحلیلگر لغوی نشان داده شده است.

```
Viod main(int arg c, char *argv[])
{
        file * intext;
        if (argc <2)
        {
        clrscr();
        textcolor(red);
        gotoxy(10,5);
        cprintf(" نام برنامه مورد کامپایل فراموش شده است");
        if (!get()) getch();
```

```
        }
        intext=fopen(argv[ 1],"R");
        while ! feof(intext)
        {
        token-type token;
        token=lexer(intext);
}
```

نقطه گنگ در مثال TokenType و در واقع نوع خروجی تحلیلگر لغوی است. معمولاً اطلاعات زیر در بسته token قرار داده می‌شود.

```
TypeDef Struct Token
        {int ROW;                       //      شماره سطر
        int COL;                        //      شماره ستون
        int BLKORD;             //      شماره آشیانه
        enum SymbolType;        //      شماره ترتیب بلاک در برگیرنده
        char Name[39];                  //      نوع لغت
} TokenType;                    //      خود لغت
```

در ساختار ارائه شده برای تعریف لغت که در اینجا بسته لغت یا token نامیده شده است شماره سطر و ستون ظاهر شده است. با استفاده از شماره سطر و ستون لغت، در هنگام صدور پیغام خطا، کامپایلر قادر به بیان دقیق مکان خطاء خواهد بود. BLKNO شماره آشیانه ای بلاک در برگیرنده یک لغت را مشخص می‌کند. تنها نام لغت برای تعیین و تشخیص آنها از یکدیگر کافی نیست بلکه، جهت تعیین یک لغت نیاز به شماره آشیانه‌ای بلاک در برگیرنده آن نیز هست. برای نمونه به قطعه کد زیر توجه کنید:

```
{
        int I;
        I=5;
        {       int I;
                I=6;
                printf('2nd blk %d ' /I );
        }
        printf('\n Its blk %d'/I);
{
```

در قطعه کد فوق همانگونه که مشاهده می‌کنید اگر چه که برای دو متغیر یک نام I تعیین شده اما شماره آشیانه‌ای بلاک در بر گیرنده، وجه تمایز این دو می‌باشد. گاهی اوقات شماره آشیانه‌ای بلاک نیز کافی نیست برای نمونه در قطعه کد زیر اگر چه که شماره آشیانه‌ای بلاک‌ها یکی است اما دو متغیر I وجود دارد.

```
{int I;
I=I+1;}
{INT I;
I=I+3}
```

شماره آشیانه ای با ورود بلاک افزایش یافته و با خروج از بلاک کاهش می‌یابد. با ظهور هر بلاک جدید شماره بلاک یا BLKORD یک واحد اضافه می‌شود، در مثال فوق وجه تمایز دو متغیر I شماره ترتیب بلاک در بر گیرنده آن است.

به هر لغت یک نوع تخصیص داده می‌شود. نوعی شمارش پذیر یا در اصطلاح Enumenated برای نمونه تحلیلگر لغوی اگر در متن فایل ورودی عدد 123 را تشکیل دهد، در داخل فیلد Name عدد 123 و در داخل فیلد Type، نوع آن یک عدد صحیح می‌باشد را مشخص می‌کند.

اگر در متن ورودی، جمله زیر قرار گرفته باشد:

ABC = 123

تحلیلگر لغوی با اجرای دستورالعملی مثل:

NextChar = getch(in-text);

ابتدا کاراکتر A و سپس B و بعد از آن C را از متن فایل ورودی خوانده، حالا با مشاهده Blank یا علامت مساوی(=) خاتمه اولین لغت را تشخیص داده، در داخل فیلد Name رشته ABC و در داخل فیلد Type، نوع آن که یک شناسه می‌باشد را قرار می‌دهد.
در فراخوانی بعدی تحلیلگر لغوی با Blank مواجه می‌شود. از آن چشم پوشی می‌کند و به جلو می‌رود و علامت = را به عنوان لغت بعدی تشخیص داده، در فیلد Name رشته "=" و در فیلد Type عملگر تخصیص را قرار می‌دهد. البته شماره سطر و ستون و بلاک در برگیرندهٔ هر لغت نیز مشخص می‌شود.
نوع لغات در فیلد Type از ساختار TokenType مشخص شده است. این فیلد از نوع enum به نام Symbols تعریف شده است. در داخل Symbol انواع ممکن لغات در داخل زبان مورد نظر مشخص می‌شود. باید توجه داشته باشید که با تعریف متغیر از نوع enum مجموعه‌ای از مقادیر ثابت یا Constant تعریف می‌شود. نوع Symbols در زبان C به صورت زیر برای نمونه تعریف می‌شود:

```
enum Symbols {S_Program, S_Const, S_Eq, S_Semi, S_Id, s_No, S_Type, S_Record, S_End,
S_Int, S_Real, S_Char, S_Array, S_String, S_Begin, S_Colon, S_ParBAZ, S_ParBast, S_Set,
S_of, S_BrackBaz, S_Var, S_BrackBast, S_Pointer, S_ConstString, S_Function, S_Procedure,
S_Begin, S_Do, S_Comma, S_If, S_Then, S_Eles, S_While, S_Do, S_Repeat, s_Until, S_For,
S_Add, S_Sub, S_Div, S_MUL, S_Mod, S_Lt, S_Le, S_Gt, S_Ge, S_Gt, S_Ne, S_Not, S_And,
S_Or};
```

همانگونه که مشاهده می‌شود انواع مختلف لغات باید در نوع Symbols گنجانده شود.

۲-۳- عبارات با قاعده

عبارات باقاعده، فرمی است چکیده و خلاصه برای بیان قوانین لغوی زبانها. شکل لغات در زبانهای برنامه نویسی دارای فرم کلی خاص است. براساس این فرم کلی است که انواع لغات از قبیل شناسه‌ها (اسامی) و اعداد، رشته‌های ثابت کاراکتری، توضیحات(Comment) و سایر لغات از یکدیگر تفکیک و تمیز داده می‌شوند.

۲-۳-۱- نمونه‌هایی از عبارات با قاعده

به عنوان نمونه تعریف شناسه‌ها در زبان c را در نظر بگیرید. یک شناسه یا Identifire در زبان c حتماً باید با یک کاراکتر آغاز گردد و در ادامه ممکن است به هر تعداد و با هر ترکیبی از ارقام صفر تا نه (0-9)و الفبای انگلیسی (A-Z) و علامت خط زیر (UnderLine)

ادامه یابد. به عنوان مثال اسامی:

A, B__1, BAC2345__

از لحاظ زبان C به عنوان اسامی(شناسه)، شناخته می‌شوند. می‌توان با استفاده از یک عبارت با قاعده به صورت زیر، فرم کلی شناسه‌ها را در زبان C تعریف نمود:

Identifire: Letter (Letter | Digit | '_')*

Letter: A | B | ……. | Z | a | b | …….| z

Digit: 0 | 1 | 2 | ……. | 9

در عبارت ارائه شده برای شناسه‌ها، علامت '*' نمایانگر تکرار صفر یا بیشتر می‌باشد و علامت '|' علامت یا می‌باشد. می‌توان به صورت چکیده‌تر نیز Digit را تعریف نمود. به همین ترتیب، حروف الفبای انگلیسی را به صورت زیر می‌توان تعریف نمود:

Digit: [0..9]

به همین ترتیب، حروف الفبای انگلیسی را به صورت زیر می‌توان تعریف نمود:

Letter: [a .. z, A .. Z]

شکل کلی اعداد صحیح را می‌توان با استفاده از یک عبارت به این صورت مشخص نمود:

Number: digit digit*

Digit: [0 .. 9]

بر طبق این عبارت، یک عدد صحیح با یک رقم بین صفر تا نه آغاز می‌شود و به هر تعدادی رقم می‌تواند در ادامه آن ظاهر شود، برای مثال:

0, 5, 0123

همانگونه که مشخص است، یک عدد حداقل دارای حداقل یک رقم باید باشد یا به عبارت دیگر یک عدد از یک یا بیشتر ارقام تشکیل شده، می‌توان عدد را به صورت زیر تعریف کرد:

Number: digit+

در اینجا علامت '+' نمایانگر تکرار یک یا بیشتر می‌باشد، یادآوری می‌کنیم که علامت ستاره '*' نمایانگر تکرار صفر یا بیشتر است. اعداد می‌توانند دارای علامت و یا بدون علامت باشند.

Number: (+ | - | λ) digit+

در این عبارت با قاعده، علامت لامبدا(λ) نمایانگر عدم وجود و یا تهی می‌باشد. عبارت فوق به این صورت خوانده می‌شود: یک عدد دارای علامت به علاوه یا منها و یا ممکن است اصلاً علامتی نداشته باشد و به دنبال آن به تعداد یک یا بیشتر ارقام بین صفر تا نه ظاهر گردد. اعداد را می‌توان به فرمی ساده‌تر با استفاده از این قاعده که هر عبارت اختیاری مثل r بصورت r* نمایش داده می‌شود. به صورت زیر مشخص نمود:

Number: (+ | -)* digit+

به عنوان نمونه‌ای دیگر از عبارات با قاعده، فرم کلی رشته‌ها در زبان C را در نظر بگیرید. باید توجه داشته باشید که در این زبان چنانچه در وسط رشته علامت کوتیشن ' نیاز باشد، می‌بایست آن را دو بار تکرار نمود. برای نمونه چنانچه جملهٔ This is your 's در خروجی مورد نظر باشد، می‌توان دستور printf("This is your""s"); استفاده نمود. لذا، فرم کلی رشته‌ها با یک عبارت با قاعده به صورت زیر بیان می‌شود:

String: "(Characters – ") * "

در عبارت فوق، مجموعه Characters نمایانگر هر گونه کرکتر قابل مشاهده منهای کوتیشن است.

۲-۳-۲- قوانین ایجاد عبارت باقاعده

برای ایجاد یک عبارت باقاعده، تعدادی از علائم مورد استفاده واقع می‌شوند. هر یک از این علائم دارای معنی و مفهوم خاصی می‌باشند. در حالت کلی اگر دلتا (Δ) مجموعه علائم بکار رفته شده در عبارات باشد، آنگاه:

۱. هر عنصر a ϵ Δ خود یک عبارت با قاعده است. برای مثال چنانچه Δ = [0 .. 9] باشد آنگاه هر رقمی بین صفر تا نه مثل ۱، خود به تنهایی یک عبارت باقاعده است.

۲. چنانچه r و s دو عبارت با قاعده باشند، آنگاه rs نیز یک عبارت با قاعده است. به عبارت ساده‌تر در حالت کلی اگر دو عبارت با قاعده را در کنار یکدیگر قرار دهید، حاصل یک عبارت باقاعده خواهد بود.

۳. چنانچه r و s دو عبارت با قاعده باشند، آنگاه r یا s که به صورت (r|s) مشخص می‌شود نیز یک عبارت باقاعده است. عملگر | دارای خاصیت جابجایی است. به عبارت دیگر:

r | s = s | r = (r | s)

۴. چنانچه r یک عبارت با قاعده باشد آنگاه r* نمایانگر تکرار صفر یا بیشتر از عبارت r است. برای نمونه اگر a، r یا b باشد آنگاه r* برابر است با:

r = a | b

r* = (a| b)*

این عبارت گویای هر ترکیبی با هر تعدادی از a یا b که در کنار یکدیگر قرار گرفته‌اند می‌باشد. برای نمونه رشته‌های aaa , babbb , bbbb همگی فرم‌های خاصی از عبارت r* هستند. بنابراین علامت لامبدا λکه نمایانگر عنصر تهی است، به تنهایی یک عبارت باقاعده است.

۵. چنانچه r یک عبارت باقاعده باشد، آنگاه r^+ نمایانگر تکرار یک یا بیشتر عبارت r است. برای نمونه :

Number: $digit^+$

به این ترتیب واضح است که:

$r^* = (r^+ | \lambda)$

با استفاده از پنج قاعدهٔ فوق، می‌توان عبارت با قاعده را بنا نمود. با استفاده از نکات فوق، فرم کلی جملات تفسیری(Comment) در زبان C را می‌توان به صورت زیر معین نمود.

Comment: Comment | Comment2

Comment1: // A^* Comment2: BC^*D A^*: هر ترکیبی از کاراکترها

B:/* D:*/ C^* : هر ترکیبی از کاراکترها بجز */

با توجه به اینکه اعداد یا به صورت صحیح و یا به صورت اعشاری ظاهر می‌شوند و با در نظر گرفتن اینکه اگر قبل از ممیز حداقل یک

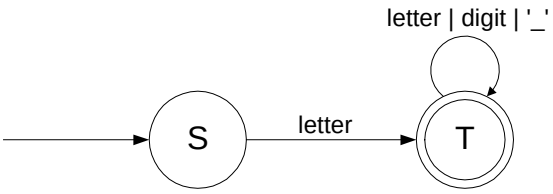
رقم ظاهر شود آنگاه وجود ارقام پس از ممیز ضروری نیست و نیز اینکه بعد از ممیز حداقل یک رقم ظاهر شود وجود رقم قبل از ممیز ضروری نیست. به طور خلاصه عبارت با قاعده برای بیان شکل کلی اعداد به صورت زیر می‌تواند باشد:

Number: $(digit^*.digit^+) \mid (digit^+.digit^*) \mid digit^+$

عبارات با قاعده مبین قوانین لغوی و نمایانگر فرم کلی لغات هستند. متأسفانه، این فرم کلی را به سادگی نمی‌توان تبدیل به کد برنامه نمود. لذا برای رفع این مشکل از ماشینهای خودکار استفاده می‌شود.

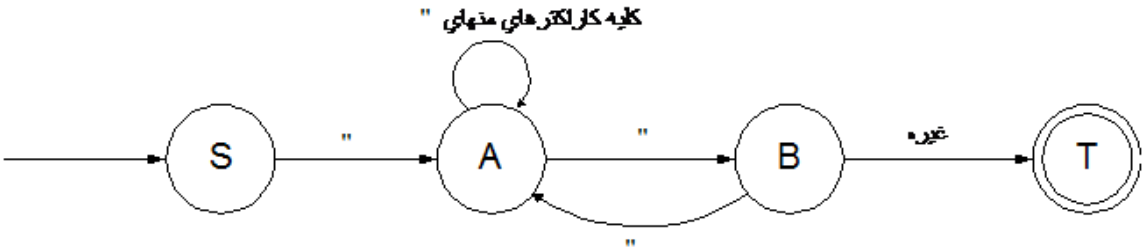
۲-۵- ماشین‌های خودکار

یک ماشین خودکار نوعی گراف می‌باشد که دارای تعدادی رئوس یا حالات (States) و تعدادی لبه یا یال (Edges) می‌باشد و در این گراف یال‌ها خطوط واصل بین حالات هستند. این ماشین‌ها، ابزاری برای تعیین قوانین لغوی و تشخیص لغات می‌باشند که به سادگی قابل تبدیل به کد برنامه بوده، به طوری که با استفاده از آنها می‌توان تحلیلگر لغوی را به صورت یک برنامه تبدیل کرد و یا Source یک برنامه را تولید نمود. برای نمونه شناسه‌ها را در نظر بگیرید. شناسه‌ها توسط عبارت باقاعده letter (letter | digit)* در بخش قبل معین شدند. برای تشخیص شناسه‌ها می‌توان ماشین خودکار زیر را مورد استفاده قرار داد:

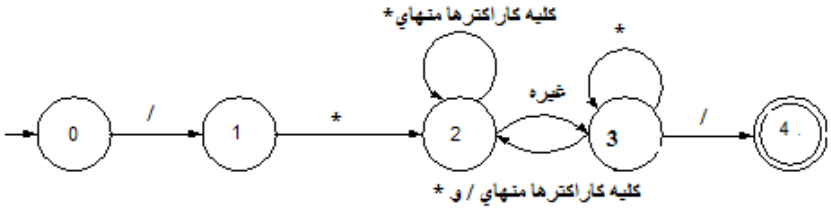


بنابر دیاگرام فوق در صورتی که کاراکتر خوانده شده از متن فایل ورودی، یکی از حروف الفبای لاتین یا در اصطلاح letter باشد، می‌توان وارد ماشین خودکار تشخیص شناسه‌ها شد. حالت شروع و در واقع نقطه ورود به ماشین خودکار با حرف S مشخص شده است. حالت بعدی T نامیده شده است. در این حالت اگر کاراکتر خوانده شده از فایل ورودی، از نوع Letter یا Digit و یا کاراکتر '-' باشد به حالت پذیرش یعنی T می‌رسد.

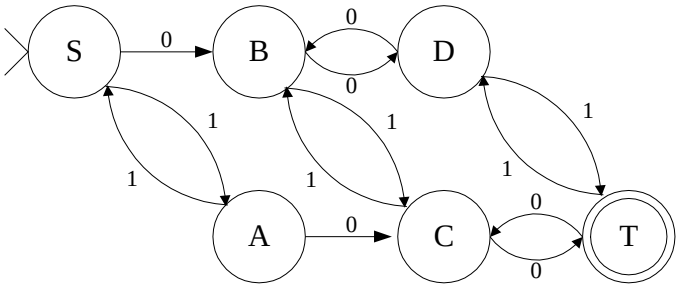
ماشین خودکار زیر هم نشان دهنده رشته‌های کاراکتری در زبان C می‌باشد.



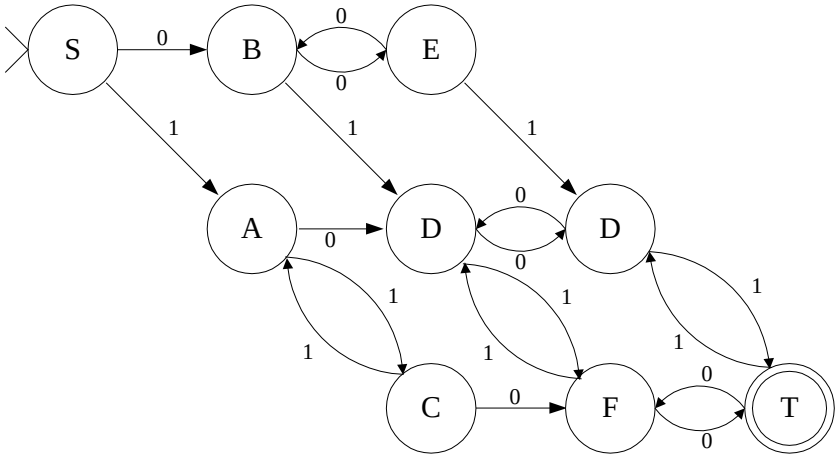
ماشین خودکار زیر نشان دهنده توضیحات در زبان C می‌باشد.



مثال: یک ماشین خودکار قطعی برای کلیه اعداد مبنای دو که تعداد صفرهای آن زوج و تعداد یک‌های آن فرد است، ایجاد کنید. در ضمن لااقل دو صفر و یک عدد یک در این اعداد موجود باشند.



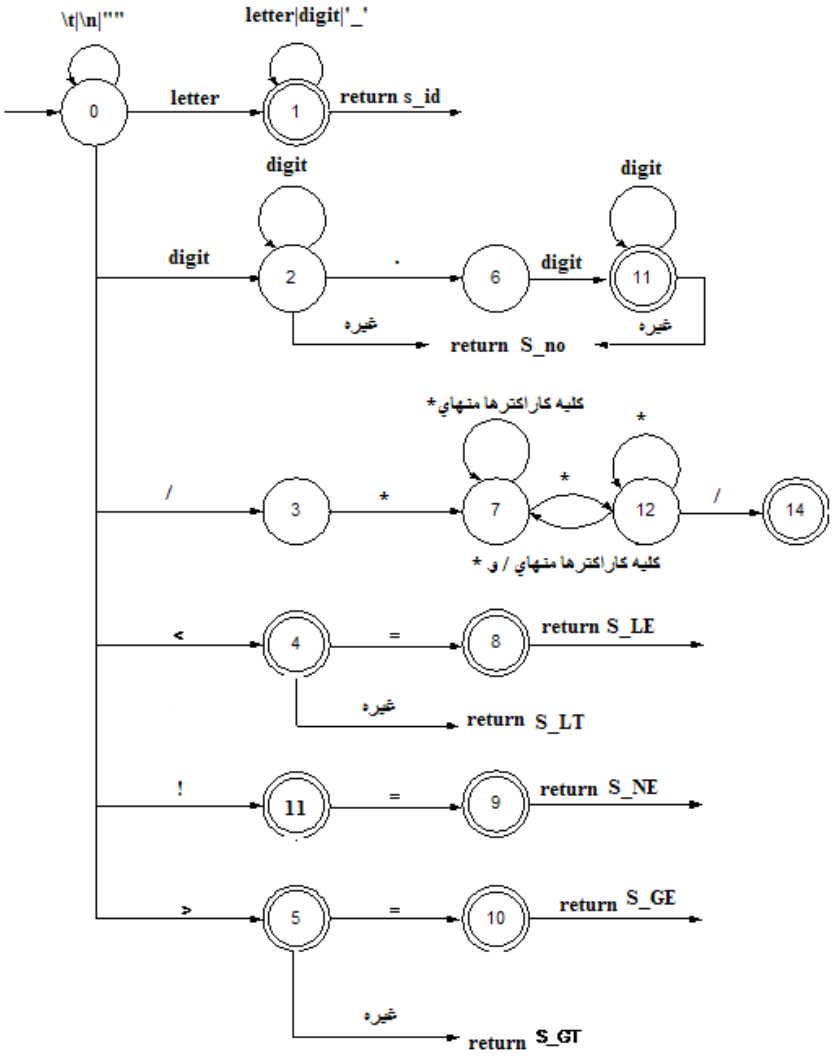
مثال: یک ماشین خودکار قطعی برای کلیه اعداد مبنای دو که تعداد صفرهای آن زوج و تعداد یک‌های آن زوج است، ایجاد کنید. در ضمن لااقل دو صفر و دو عدد یک در این اعداد موجود باشند.



۲-۶- ایجاد تابع تحلیلگر لغوی

همانگونه که در بخش ۲-۵ ذکر شد، می‌توان با استفاده از ماشین‌های خودکار قوانین لغوی را بیان نمود. در این بخش نشان داده خواهد شد که چگونه می‌توان ماشین خودکار را به سادگی تبدیل به کد برنامه نمود. برای این منظور یک دیاگرام کلی برای نمایش برخی از لغات ارائه می‌شود. دیاگرام ارائه شده در شکل زیر نمایانگر ماشین خودکار کلی برای تشخیص تعدادی از لغات است. در واقع این دیاگرام بیان کننده الگوریتم تابع تحلیلگر لغوی است.

هرگاه تابع تحلیلگر لغوی مورد فراخوانی قرار گردد، در حالت شروع صفر قرار می‌گیرد. اگر در فراخوانی قبل، کاراکتری اضافه خوانده شده بود، آن کاراکتر اضافه را مورد استفاده قرار می‌دهد، وگرنه کاراکتر بعدی را از داخل متن برنامه مورد کامپایل می‌خواند. تا زمانی که کاراکترهایی که ارزش لغوی ندارند، مانند NewLine، Tab و Blank خوانده شوند، تحلیلگر لغوی در همان حالت شروع صفر باقی می‌ماند. در غیر اینصورت وابسته به نوع کاراکتر، در یک جمله Case اقدام به تشخیص لغات مختلف می‌نماید. می‌توان با استفاده از دیاگرام شکل بالا، کد تحلیلگر لغوی را به شرح زیر در زبان C++ ایجاد نمود.




```
class TokenType lexer (FILE *InText)
{
        enum Symbols LexiconType;
        char NextChar, NextWord[80];
        int State, Length;
        static char LastChar = '\0';
        static int RowNo =0, ColNo= 0;
        State= 0;                                   در وضعیت شروع قرار می‌گیرد//
        Length= 0;
        While (!Eof(InText))
        {
                if (LastChar)
                {NextChar= LastChar; LastChar= '\0';}
                else NextChar= fgetc(InText);
```

```
        NextWord[Length++]= NextChar;
        Switch (State)
        {
    case 0:                         //حالت شروع صفر
    if (NextChar== '\n') {RowNo++; ColNo= 0;}
    else ColNo++;
    if (NextChar== ' ' | NexrChar== '\t' | NextChar=='\n') Length= 0;
    else if ((NextCHar<= 'z' && NextChar>= 'a')||(NextChar<= 'Z' && NextChar>= 'A'))
                                                                    State= 1;
    else if (NextChar<= '9' && NextChar>= '0') State =2;
    else if (NextChar== '/') State= 3;
    else if (NextChar== '<') State= 4;
    else if (NextChar== '>') State= 5;
    else if (NextChar== '!') State= 11;
    else LexerError (NextWord, Length);
    break;                          //پایان حالت صفر
    case 1:                         //تشخیص شناسه‌ها
     if ((NextCHar<= 'z' && NextChar>= 'a') || (NextChar<= 'Z' && NextChar>= 'A')
            || (NextChar<= '9' && NextChar>= '0')||(NextChar=='_')) ColNo++;
    NextWord[Length-1] ='\0';
    Return MakeToken(IsKeyWord(NextWord),RowNo,ColNo);
    Break;
    Case 3:                             //تشخیص اعداد
    .
    .
    .
    }                                   //پایان سویچ
}                                       //پایان حلقه
```

در مثال فوق تابع MakeToken کار ایجاد بسته لغات یا در اصطلاح Token را برعهده دارد. این تابع در زبان C به شرح زیر است:

```
enum Symbols IsKeyWord(char *key)
{
    in I;
    struct KeyType
    {char *key; enum SymbolsType}
    KeyTab[]= {'if', S_If, 'while', S_While, 'then',
```

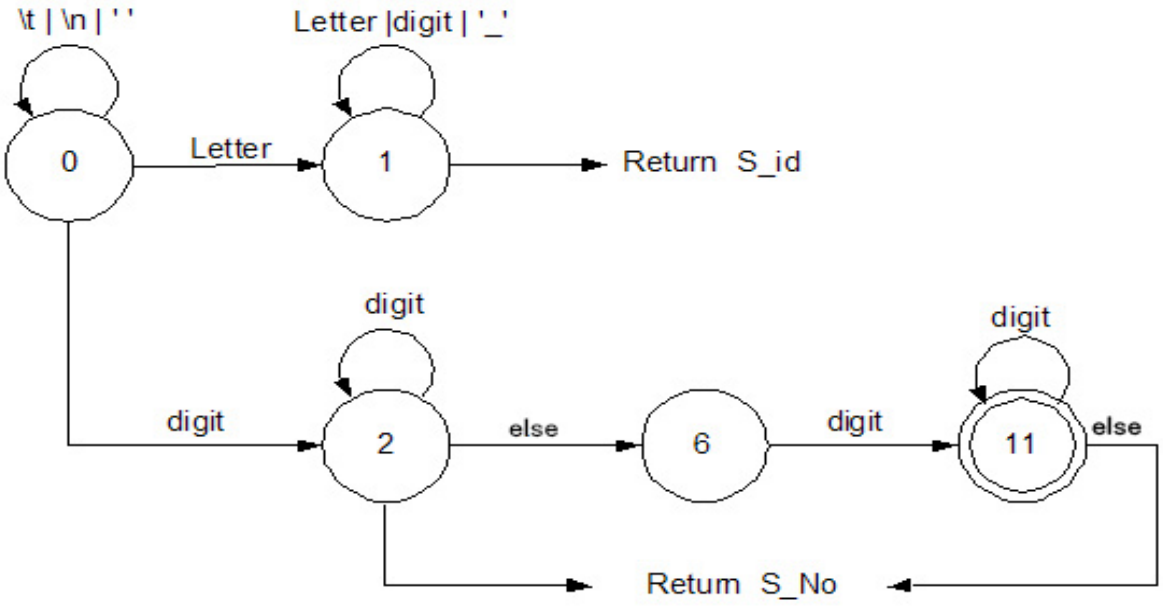
```
S_Then, ‘else’, S_Else, ‘integer’, S_Integer,
‘type’, S_Type, ‘function’, S_Function, ..}
for (I= 0; KeyTab[I].key &&
strcmp(KeyTab[I].key, key); I++);
if (KeyTab[I].key) return KeyTab[I].Type;
Return S_Identifier;
} //EOF IsKeyWord
```

۲-۷- ایجاد مولد تحلیلگر لغوی

می‌توان به سادگی تابع مولد تحلیلگر لغوی ایجاد نمود. کافیست، تابعی برای پیمایش یک گراف بنویسید. این تابع در ورودی خود علاوه بر متن برنامه مورد تحلیل لغوی، فایل قوانین لغوی را در فرم یک ماشین خودکار می‌پذیرد.

از مولدهای شناخته شده تحلیلگر لغوی، یکی LEX می‌باشد. این مولد امروزه بخصوص بر روی سیستمهای عامل UNIX موجود می‌باشد. کار با LEX مشکل است چرا که نیاز به بیان قوانین لغوی، در فرم عبارات باقاعده دارد. در صورتی که همانگونه که تا کنون مشاهده کرده‌اید، به فرمی خیلی ساده‌تر می‌توان با استفاده از ماشینهای خودکار، قوانین لغوی را بیان نمود.

باید قوانین لغوی را در قالب یک ماتریس مشخص نموده و در داخل یک فایل متن قرار داد. سپس در داخل برنامه می‌توان به این فایل ارجاع و ماشین خودکار را در قالب یک ماتریس مشخص نمود. برای نمونه در زیر قوانین لغوی در قالب ماشین خودکار و ماتریسی مشخص شده است.



0, \t, 0	1, letter, 1	2, digit, 2
0, \n, 0	1, digit, 1	2, 6
0, ‘‘, 0	1, -, 1	2, Else, AccState_ID
0, letter, 1	1, Else, AcceptState_ID	6, digit, 11
0, digit, 2		11, digit, 11
		11, Else, AccState_NO

برای ایجاد یک ماتریس نیاز به یک زبان ساده است که بر اساس آن برای نمونه بتوان مشخص نمود که برای مثال Letter چیست و یا غیره، چه می‌باشد. شاید بهتر بود که در فایل ورودی در داخل ماتریس قوانین لغوی Letter به صورت زیر مشخص می‌شد:

0, [A..Z, a..z], 1

حالا در داخل برنامه، این ساختار را به عنوان یک فایل text، کرکتر به کرکتر خوانده، عیناً در داخل یک ماتریس با تعداد سطرها که مساوی با تعداد رکوردهای فایل است و تعداد ستون‌ها که مساوی با طول هر رکورد یا هر سطر است، ضبط نمود.

اکنون با نوشتن یک برنامه پیمایش کننده ماشین خودکار و یا تولید متن یک برنامه C که بر اساس این ماشین خودکار برای انجام عمل تحلیل لغوی ایجاد شده، می‌توان لغات را تشخیص داد. همزمان با خواندن هر کرکتر از ورودی، مثلاً با خواندن کرکتر کوچکتر از ورودی طبق ماتریس از حالت شروع صفر به حالت ۵ تغییر وضعیت داده می‌شود. در اینجا سرعت عملیات تحلیلگر لغوی، وابسته به سرعت جستجو در داخل ماتریس است.

۲-۸- تبدیل برنامه‌ها به زبان فارسی

می‌توان با ایجاد یک تحلیلگر لغوی برای تشخیص متن برنامه‌ها با لغات کلیدی فارسی اقدام نمود. برای مثال جملهٔ زیر را می‌توان به یک جمله if تبدیل نمود:

اگر حقوق > ۱۰۰۰ آنگاه مالیات = حقوق / ۱۰۰ ؛

می‌توان متن برنامه‌های فارسی را که بر اساس قواعد زبانهای متفاوت نوشته شده است را با استفاده از یک جدول تبدیل کلی به زبان انگلیسی تبدیل کرد و بالعکس. پس از کامپایل و اشکال زدایی برنامه‌ها دوباره با استفاده از همان جدول از انگلیسی به فارسی تبدیل نمود.

اصولاً زبانهای سطح بالا به خاطر خوانا بودنشان، سطح بالا خوانده می‌شوند. متأسفانه این انگلیسی بودن زبانها، خوانایی برنامه‌ها را با مشکل مواجه نموده است. به خصوص در انتهای مراحل تجزیه و تحلیل سیستم‌ها، برنامه سازان را در تبدیل متن عملیات تحلیل شده از شبه دستور العمل یا در اصطلاح Pseudo Code به کد برنامه عملاً با مشکل زیاد روبرو نموده است. شبه دستورالعمل را فارسی زبان‌ها نمی‌توانند به سادگی زبان انگلیسی بنویسند. شبه دستورالعمل‌ها معمولاً از زبان‌های سطح بالا الهام گرفته می‌شوند. مشکل زبان فارسی از راست به چپ بودن جملات و بالعکس، چپ به راست بودن عبارات است. البته، در اینجا این مشکل موردی ندارد.

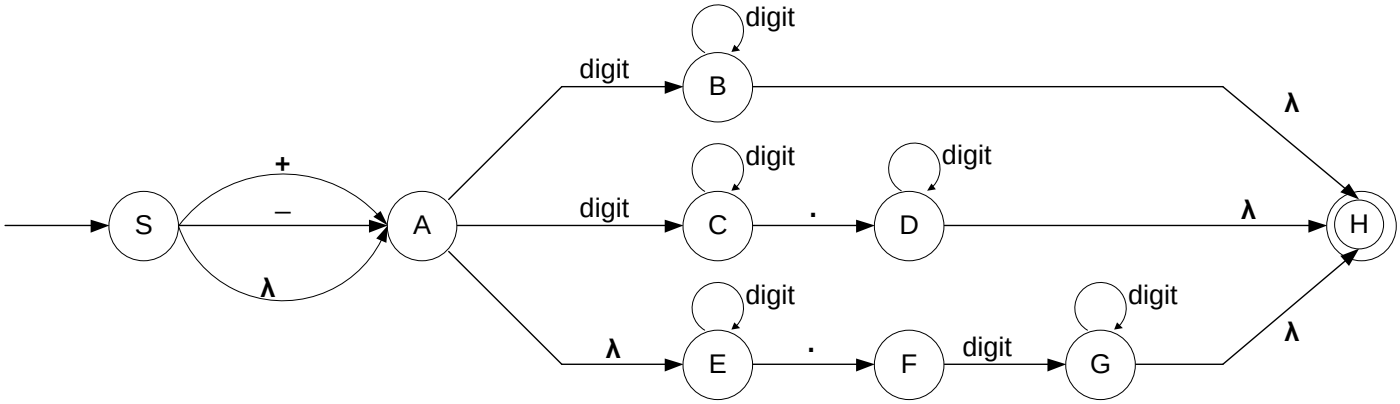
در مورد شناسه‌ها نیز در زبان فارسی مشکل وجود دارد. شناسه‌ها از انتها تشخیص داده می‌شوند. مشکل دیگر این است که در زبان انگلیسی ۲۸ و در زبان فارسی ۳۲ حرف وجود دارد. البته، با ترکیب حروف می‌توان این مشکل را حل نمود.

نکته دیگر، محیط ویرایش متن برنامه‌هاست. در اغلب محیط‌های ارائه شده برای کامپایلرها، Syntax Directed Editors ویرایشگرهایی می‌باشند که بنابر قوانین زبان به برنامه‌ساز کمک می‌کنند تا بتوانند برنامه خود را با Syntax صحیح ایجاد کنند. می‌توان یک محیط قوی ویرایش وابسته به زبان مورد نظر برای ویرایش برنامه‌های فارسی ایجاد نمود.

به این ترتیب می‌توان برنامه‌ها را به زبان فارسی با استفاده از یک ویرایشگر باهوش برای زبان خاص نوشت. سپس با استفاده از جدول تبدیل برنامه به زبان اصلی ترجمه و پس از اشکال زدایی دوباره با استفاده از جدول تبدیل، لغات را به فارسی برگرداند.

۲-۹- ماشینهای خودکار(FSA)

در اینجا ترسیم سه ماشین خودکار برای اعداد مستقل از یکدیگر ساده است. مشکل، ترکیب این سه حالت و ایجاد یک ماشین خودکار غیر قطعی است. می‌توان به سادگی و بدون در نظر گرفتن مشکل ترکیب گراف برای سه نوع متفاوت اعداد اقدام به ترسیم یک ماشین خودکار برای تشخیص سه نوع عدد نمود. زیبایی کار در اینجاست که می‌توان به صورت الگوریتمی و بدون نیاز به هیچگونه کار اضافی، این سه حالت را در قالب یک ماشین خودکار غیر قطعی با یکدیگر ادغام نمود و با استفاده از الگوریتم‌هایی که ارائه خواهد شد به طور اتوماتیک تبدیل به یک ماشین خودکار قطعی نمود.



همانگونه که در شکل بالا مشاهده می‌شود. یک ماشین خودکار غیر قطعی می‌تواند بیانگر حالات مختلف برای یک لغت باشد. در اینجا منظور از لغت، اعداد می‌باشند. اعداد یا دارای علامت هستند و یا علامت ندارند. این نداشتن علامت را با گذر تهی که با علامت λ (لامبدا) مشخص شده، معین می‌کنند. همانگونه که مشاهده می‌شود، در حالت A با دیدن digit نمی‌توان مشخص نمود که حالت بعدی آیا حالت B یا C و یا حالت E است. باید توجه داشته باشید که گذر λ در واقع یعنی هیچ چیز.

برای تبدیل ماشین خودکار غیر قطعی به قطعی در سه مرحله عمل می‌شود:

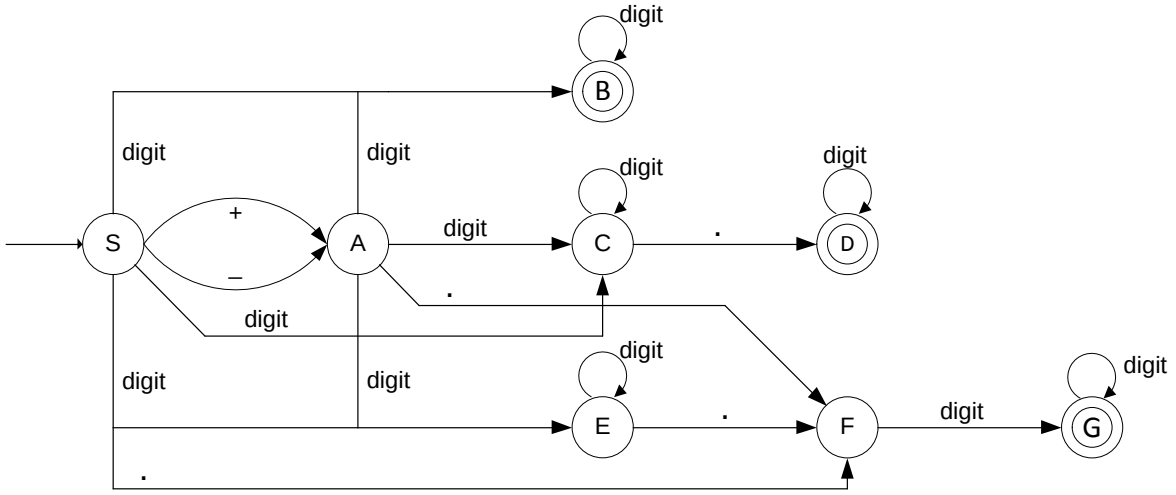
۱- حذف گذرهای تهی

۲- رفع عدم قطعیت

۳- بهینه سازی ماشین خودکار

۲-۹-۱- حذف گذرهای تهی

همانگونه که مشاهده می‌شود، کلیه واکنشهای حالت یک و یا به عبارت دیگر کلیه گذرهای خارج شونده از حالت یک، برای حالت صفر در نظر گرفته شد. به این ترتیب گذر تهی حذف گردید. در مرحله بعد به همین ترتیب ادامه می‌دهیم تا کلیه گذرهای تهی را از داخل ماشین خودکار قطع نماییم.

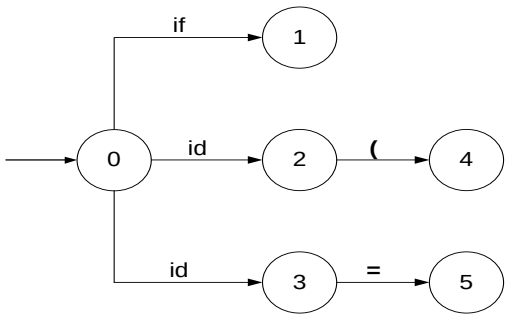


۲-۹-۲- رفع عدم قطعیت

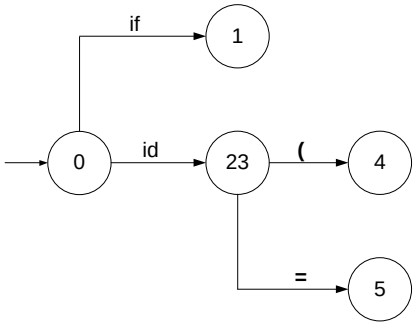
همانگونه که در شکل بالا مشاهده می‌کنید، ماشین خودکار غیر قطعی است. برای نمونه در حالت A با دیدن digit نمی‌توان تشخیص داد که آیا حالت بعدی C ، B و یا اینکه E می‌باشد. اما نکته جالب توجه اینجاست که با دیدن digit حتماً به یکی از سه حالت گذر می‌شود. برای روشن‌تر شدن روش حذف عدم قطعیت بهتر است که کار تحلیلگر نحوی مطرح شود.

تحلیلگر نحوی، معمولاً با دیدن یک لغت در ورودی، ساختار جمله مورد نظر را می‌تواند پیش‌بینی کند. برای مثال اگر لغت دریافتی از تحلیلگر لغوی if باشد، تحلیلگر نحوی بلافاصله تصمیم می‌گیرد که جمله باید جمله if باشد و باید انتظار دیدن شرط جمله if را در

ورودی داشته باشد. اما، یا به عبارت دیگر یک شناسه، مشکل وجود دارد. در اینجا تحلیلگر نحوی نمی‌تواند مشخص کند که آیا جمله مورد نظر یک جمله فراخوانی زیر برنامه‌ها و یا یک جمله تخصیصی (Assignment) است. ماشین خودکار غیرقطعی در اینجا به صورت زیر قابل ترسیم است:



علیرغم وجود عدم قطعیت در حالت صفر از شکل بالا می‌توان قطعیتی را در نظر داشت. به این ترتیب که در حالت شروع صفر اگر یک شناسه یا id در ورودی ظاهر شود، قطعاً حالت بعدی، حالت ۲ یا ۳ خواهد بود. در حالت ۲ یا۳ اگر '=:' در ورودی ظاهر شود جمله تخصیصی، اگر ')' ظاهر شود، جمله فراخوانی و در غیر اینصورت جمله غلط می‌باشد. پس حالات ۲ یا ۳ را به صورت یک حالت ترکیبی جدید با ادغام خروجی‌های این دو حالت می‌توان به وجود آورد.



برای سهولت در امر تبدیل و یا نمایش ماشینهای خودکار به جای گراف از فرم جدول مانند استفاده می‌شود. برای نمونه ماشین خودکار غیر قطعی اعداد در سه شکل قبل به صورت یک جدول در شکل زیر مشخص شده است.

	Digit	.	- / +
S	BCE	F	A
A	BCE	F	
B	B		
C	C	D	
D	D		
E	E	F	
F	G		
G	G		

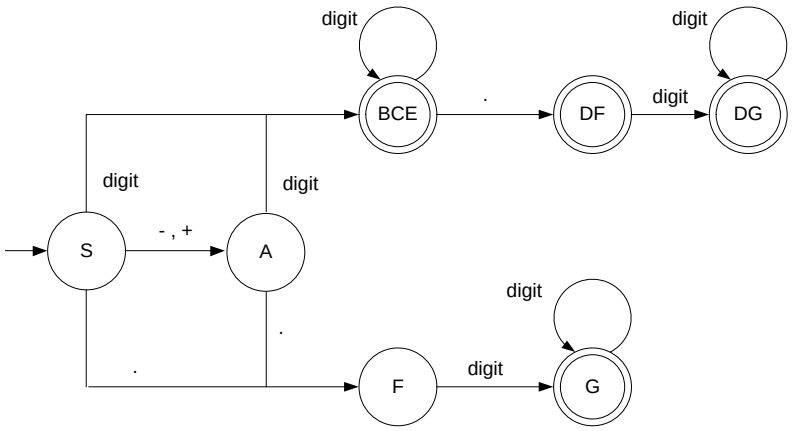
همانگونه که در شکل بالا مشاهده می‌شود در حالات S و A به یکی از سه حالت B یا C یا E به ازای دیدن digit گذری وجود دارد. لذا، می‌توان گفت که گذری به حالت ترکیبی BCE وجود دارد. در این حالت می‌توان واکنشهای هر سه حالت C، B و E را داشت. ردیف مربوط به حالت ترکیبی BCE در شکل زیر ترکیبی از واکنشهای هر سه حالت تشکیل دهنده آن است به عبارت دیگر از ترکیب 'یا' سطرهای B، C و E، سطر جدید BCE به ماتریس افزوده می‌شود. BCE یک حالت پذیرشی است، زیرا یکی از حالات تشکیل

دهنده آن یعنی B یک حالت پذیرش است.

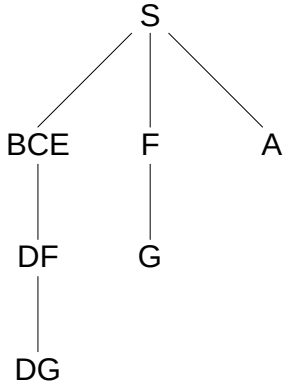
همانگونه که در شکل زیر مشاهده می‌شود. در حالت جدید BCE به ازای ورودی '.' (نقطه اعشار) عدم قطعیت وجود دارد. از حالت BCE می‌توان به هر یک از دو حالت D یا F گذر نمود. لذا برای عدم قطعیت، حالت جدید دیگری به نام DF را به جدول حالات باید افزود. برای رفع عدم قطعیت در حالت DF حالت DG ایجاد می‌شود.

	Digit	.	- / +
S	BCE	F	A
A	BCE	F	
B	B		
C	C	D	
D	D		
E	E	F	
F	G		
G	G		
BCE	BCE	DF	
DF	DG		
DG	DG		

اکنون با استفاده از جدول فوق می‌توان ماشین خودکار قطعی اعداد را به صورت زیر ایجاد نمود.



همانگونه که مشاهده می‌کنید برخی از حالات درون جدول مثل حالات B, C, D و E در عمل غیرقابل دسترسی هستند. علت افزایش حالات جدید ترکیبی است. می‌توان این حالات زاید را با ایجاد درخت دسترسی نیز مشخص نمود.



با افزایش حالات جدید جهت عدم قطعیت در ماشین خودکار، برخی از حالات زاید شده، از حالت شروع غیر قابل دسترسی می‌شوند. جهت تشخیص این حالات یا می‌توان ماشین خودکار را از روی جدول ترسیم نمود. و یا اینکه با استفاده از یک درخت دسترسی حالات قابل دسترسی از حالت شروع S را مشخص نمود. برای نمونه درخت دسترسی برای ماتریس دو شکل قبل به صورت زیر است. باید توجه داشته باشید که نام حالات در درخت دسترسی تکراری نیست و این درخت صرفاً جهت تعیین حالات قابل دسترسی از حالت شروع S است.

۲-۹-۳- بهینه‌سازی ماشین‌های خودکار

هدف از بهینه سازی، تقلیل تعداد حالات در ماشین‌های خودکار است. برای این منظور باید حالات معادل را تشخیص داد. دو حالت را در صورتی معادل گویند که به ازای ورودی‌های متفاوت با گذشت از یک سری از حالات یا به عبارت دیگر در مسیری به طول صفر یا بیشتر و با گذشتن از تعداد n گره نهایتاً به حالت پذیرش برسند. حالات معادل را با روشی ابتکاری به ترتیب زیر می‌توان مشخص نمود:

۱. ابتدا حالات معادل صفر که با مسیرهایی به طول صفر به پذیرش می‌رسند را از سایر حالات مجزا باید نمود. به عبارت دیگر، حالات پذیرش از حالات غیر پذیرش مجزا می‌شوند. به این ترتیب حالات به دو دسته مجزا از حالات پذیرش و غیر پذیرش تقسیم می‌شوند.

۲. به درون هر دسته حالات باید نگریست و مشخص نمود که آیا ورودی وجود دارد که به ازای آن حالات متعلق به یک دسته واکنش‌ها جدا از سایرین دارند. واکنش‌ها بر اساس گذر یک حالت به حالت دیگر سنجیده نمی‌شوند. بلکه، اگر از یک حالت به حالت دیگر گذری وجود دارد، دسته مربوط به آن حالت را مشخص می‌کنند. مقصود مقایسه دسته‌هایی است که به آنها گذر می‌شود. حالات مشابه به ازا یک ورودی به حالات درون یک دسته گذر می‌کنند.

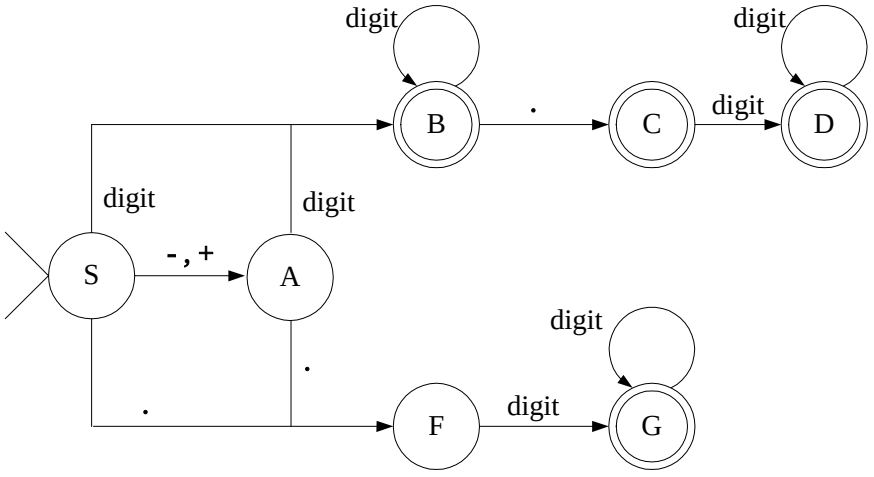
۳. آنقدر مرحلهٔ ۲ تکرار می‌شود تا دیگر گذری متفاوت بین دسته‌ها وجود نداشته باشد و دسته‌ای جدیدتر تولید نشود. برای نمونه اکنون ماشین خودکار که در مبحث قبل مطرح شد بهینه‌سازی می‌شود.

به طور خلاصه و با در نظر گرفتن الگوریتم فوق می‌توان حالات معادل را به صورت زیر تعریف نمود:

دو حالت را در صورتی معادل k گویند که به ازای هر رشته از ورودی‌ها به طول کوچکتر یا مساوی با k اگر از یکی از آنها بتوان به پذیرش رسید، از دیگری نیز بتوان به ازای همان رشته به حالت پذیرش رسید.

دو حالت را معادل گویند اگر و تنها فقط اگر به ازای هر رشته‌ای از ورودی‌های متوالی که موجب رسیدن از آن حالت به یک حالت پذیرش است بتوان از دیگری نیز به پذیرش رسید.

اکنون با توجه به تعریف حالت معادل بهینه‌سازی ماشین‌های خودکار، می‌توان مبادرت به بهینه‌سازی ماشین خودکار اعداد که در مبحث قبل مطرح شد، نمود. ماشین خودکار قطعی اعداد و جدول تولید آن در شکل بخش مربوطه ارائه شده است.



	Digit	.	- / +
S	B	F	A
A	B	F	
B	B	C	
C	D		
D	D		
F	G		
G	G		

برای بهینه‌سازی ماشین‌های خودکار ابتدا باید گره‌ها را به دو دسته پذیرشی و غیر پذیرشی افراز نمود. به عبارت دیگر حالت معادل صفر که با مسیرهایی به طول صفر به پایان می‌رسند را از سایر حالات باید تفکیک نمود. به این ترتیب حالات ماشین خودکار اعداد که در شکل‌های بالا ارایه شده است به دو دسته زیر تقسیم می‌شوند:

۱- حالات پذیرشی {B, C, D.G}

۲- حالات غیرپذیرشی {S, A, F}

حال باید مشخص نمود که آیا در داخل هر دسته به ازای هر ورودی، گره‌ها واکنشی یکسان دارند یا خیر. منظور از واکنش یکسان این است که برای نمونه اگر از حالت M در دسته شماره ۱۲ به ازای ورودی Digit گذری به حالتی در دسته شماره ۷ وجود دارد یا خیر. سایر حالات موجود در دسته شماره ۱۲ باید به ازای ورودی Digit گذری به حالتی در دسته شماره ۷ داشته باشند. در غیر اینصورت، این حالات از داخل دسته خارج شده و در دسته‌ای جدید قرار می‌گیرند.

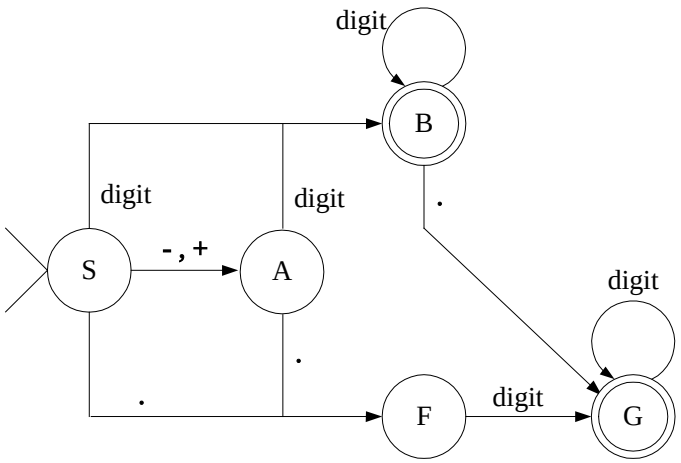
برای نمونه در داخل دسته شماره ۱ در بالا به ازای ورودی نقطه اعشار '.' از حالت B یک خروجی و گذری وجود دارد در صورتی‌که در مورد سایر حالات این چنین نیست. پس این حالت از سایرین جدا می‌شود و سه دسته به صورت زیر حاصل می‌گردد:

۱ {C, D.G}

۲ {B}

۳ {S, A, F} حالات غیرپذیرشی

به همین ترتیب در دسته شماره ۳ حالت S قابل تفکیک از A و F است. زیرا در S به ازای ورودی‌های + و − گذر و واکنشی وجود دارد در صورتی‌که در مورد A و F اینچنین نیست. پس حالت S از A و F قابل تفکیک است. از سوی دیگر A و F نیز قابل تفکیک از یکدیگر هستند. بنابراین سه حالت C، D و G غیر قابل تفکیک از یکدیگر و معادل هستند. پس می‌توان هر یک از این سه حالت معادل را به جای دو حالت دیگر در داخل ماشین خودکار قرار داد. به این ترتیب ماشین خودکار بهینه اعداد به صورت زیر خواهد بود:

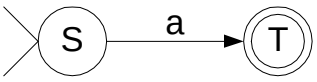


	Digit	.	- / +
S	B	F	A
A	B	F	
B	B	C	
F	G		
G	G		

۲-۱۰- تبدیل عبارات باقاعده به ماشین‌های خودکار

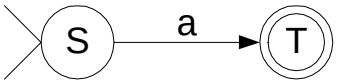
در این قسمت نشان داده خواهد شد که چگونه می‌توان عبارات باقاعده را به صورت ماشین‌های خودکار تبدیل نمود تا اینکه بتوان با استفاده از ماشین خودکار، قوانین لغوی که در فرم عبارات باقاعده خلاصه شده‌اند را عملاً به صورت کد برنامه مورد بهره برداری قرار داد. برای تبدیل عبارات باقاعده به ماشین‌های خودکار معمولاً مراحل زیر به کار می‌روند:

۱. هر عنصر a متعلق به الفبای زبان به صورت یک ماشین خودکار نمایش داده می‌شود:

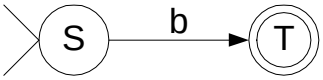


۲. ماشین خودکار برای عبارت ab را از ترکیب گراف‌ها برای a و b به صورت زیر می‌توان ایجاد نمود:

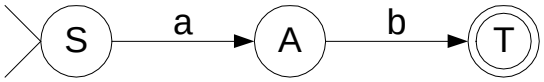
(الف) ماشین خودکار برای a



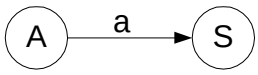
(ب) ماشین خودکار برای b

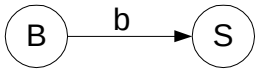


(ج) ماشین خودکار برای ab

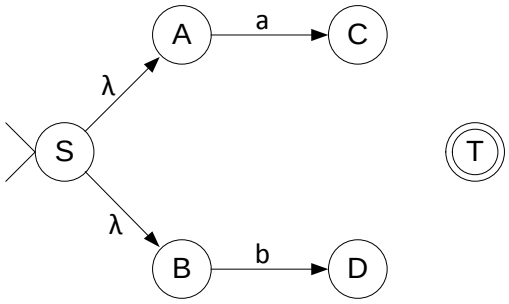


۳. ماشین خودکار برای عبارت a | b را با یک استدلال ساده می‌توان از ماشین‌های خودکار برای عبارات a و b ایجاد نمود. چنانچه حالت شروع ماشین خودکار برای عبارت a | b حالت S باشد:



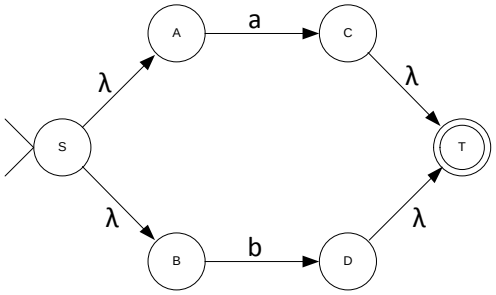


حالت شروع S هیچ فرقی با حالت شروع برای عبارت a ندارد زیرا در شروع a | b می‌توان a را هم در ورودی دید. از سوی دیگر حالت شروع S هیچ تفاوتی با حالت شروع برای عبارت b ندارد.



به همین ترتیب با مشاهده a در ورودی باید مطمئن بود که a | b در ورودی ظاهر شده است.

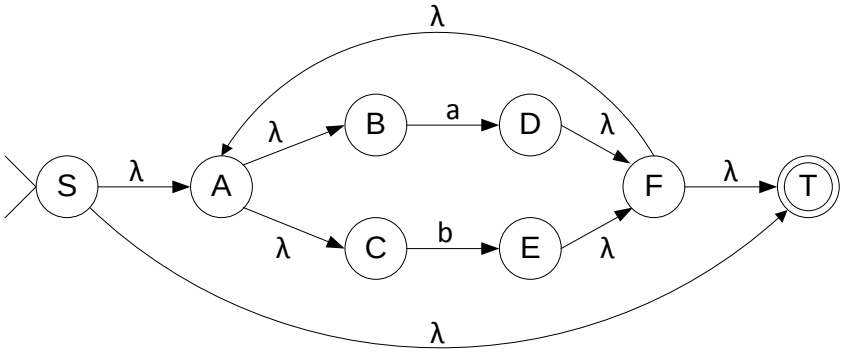
لذا حالت پذیرش برای ماشین خودکار عبارت a یعنی C هیچ تفاوتی با حالت پذیرش برای ماشین خودکار برای عبارت a | b یعنی T ندارد. به همین ترتیب حالت D هیچ تفاوتی با حالت پذیرش T ندارد. بنابراین ماشین خودکار برای عبارت a | b بصورت زیر خواهد بود:



در شکل فوق عبارت a | b از ترکیب ماشین‌های خودکار برای عبارت a و b ساخته شده است. مسلم است که حالت شروع S هیچ تفاوتی با حالت A و حالت B ندارد. اما بالعکس صادق نیست. و حالت شروع برای تشخیص a یعنی A متفاوت از حالت شروع برای a | b است. زیرا در حالت شروع برای a نمی‌توان در ورودی b را دید. لذا، حالت شروع a | b را با دو گذر تهی به حالت شروع برای a و حالت شروع برای b بایدمتصل نمود.

۴. با در دست داشتن ماشین خودکار برای a و b و در نتیجه وجود ماشین خودکار برای a | b می‌توان ماشین خودکار برای عبارت *(a | b) را با یک استدلال ساده ایجاد نمود. به این ترتیب که فرض کنید حالات شروع و خاتمه برای ماشین خودکار *(a | b) به ترتیب حالات S و T باشند. اولاً، چون *(a | b) می‌تواند تهی باشد، پس حالت شروع آن ممکن است هیچ تفاوتی با حالت پذیرش نداشته باشد. زیرا *(a | b) ممکن است اصلاً وجود نداشته باشد.

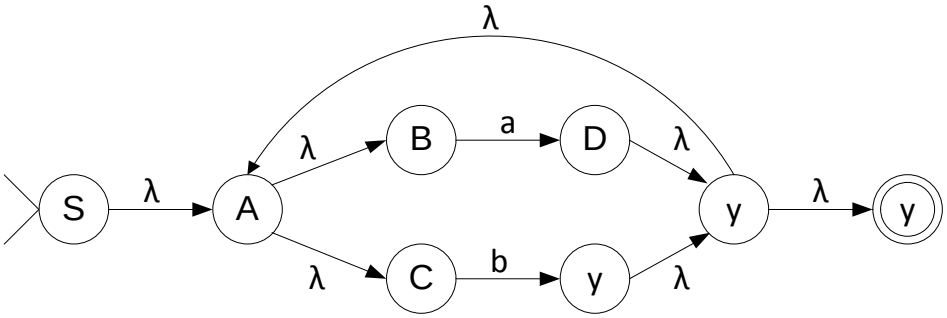
از طرف دیگر ممکن است حالت شروع S هیچ تفاوتی با حالت شروع (a | b) نداشته باشد. زیرا، در شروع *(a | b) می‌توان در شروع مشاهده a | b در ورودی بود. پس از مشاهده a | b دوباره می‌توان در آغاز مشاهده a | b جدیدتری قرار گرفت. این در واقع به خاطر



خاصیت تکراری بودن *(a | b) است. پس، حالت پذیرش a | b درون *(a | b) هیچ تفاوتی با حالت شروع آن نخواهد داشت. با این استدلال می‌توان نتیجه گرفت که ماشین خودکار برای عبارت *(a | b) به صورت زیر است.

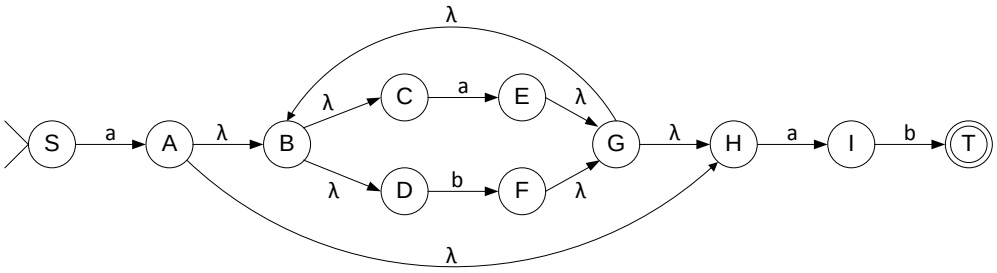
مسلماً در ورودی *(a | b) چون رشته a یا b می‌تواند اصلاً وجود نداشته باشد، در نتیجه حالت شروع می‌تواند معادل با حالت پذیرش باشد. لذا در شکل بالا، حالت شروع S با گذری تهی به حالت پذیرش T متصل شده است. از جهت دیگر در خاتمه دیدن a | b مثل اینکه دوباره در حالت شروع بوده و می‌توان a | b جدیدی را مشاهده نمود و این عمل تا به هر تعدادی قابل تکرار است. برای این منظور حالت F با گذری تهی به حالت شروع a | b یعنی A متصل شده است. علاوه بر این حالت، F ممکن است خاتمه تکرار و حالت پذیرش هم باشد.

۵. عبارت +(a | b) را می‌توان با حذف گذر تهی از حالت شروع به حالت پایان در ماشین خودکار برای *(a | b) تولید نمود، زیرا در مورد +(a | b) حداقل یکبار a | b در ورودی باید ظاهر شود و حالت شروع آن با خاتمه یکسان نیست.

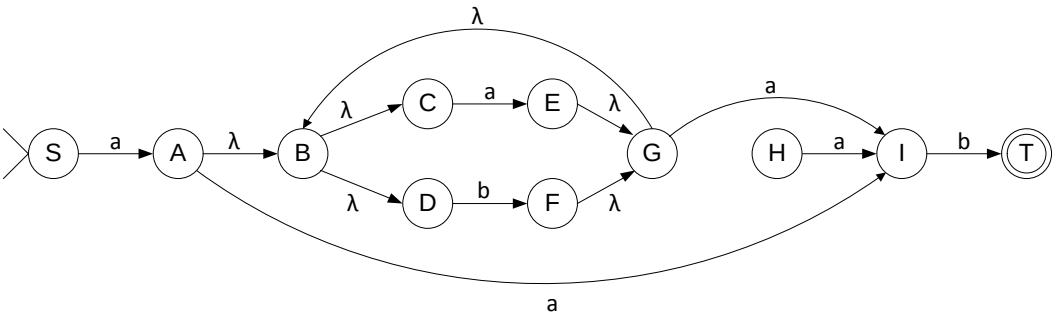


مثال ۱: برای عبارت a (a | b)* ab یک ماشین خودکار بهینه ایجاد نمایید.

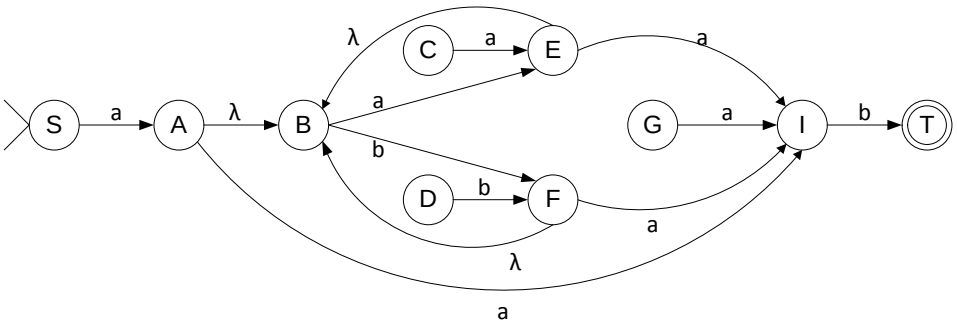
با استفاده از ماشین خودکار ارایه شده برای عبارت *(a | b) که در دو شکل قبل ارایه شده است، می‌توان به سادگی ماشین خودکار غیرقطعی را ایجاد نمود.



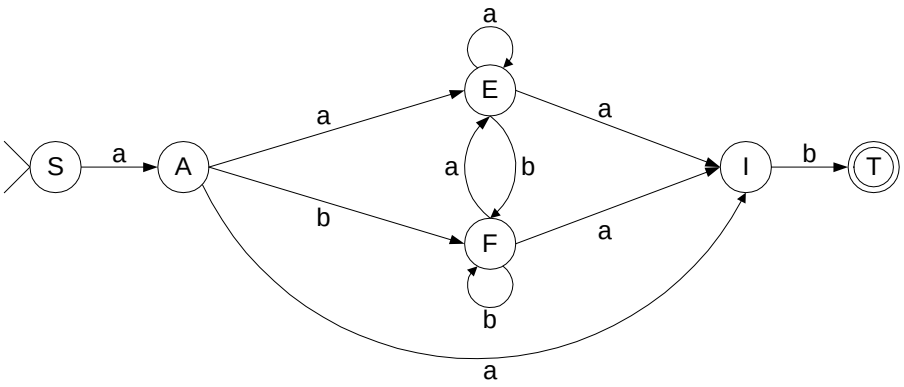
پس از حذف گذرهای تهی به H ماشین به صورت زیر تبدیل می‌شود.



مشاهده می‌کنید که حالت H در شکل بالا غیرقابل دسترسی و قابل حذف است. در شکل زیر گذرهای تهی به حالات G، C و D حذف شده‌اند.

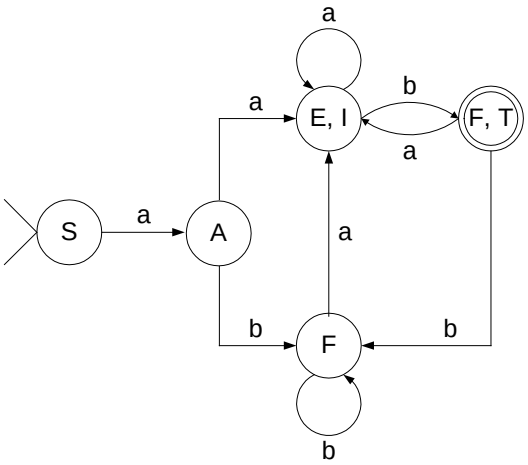


پس از حذف گذرهای تهی ماشین به صورت زیر تبدیل می‌شود.



ماشین خودکار فوق در شکل زیر با استفاده از نمایش ماتریسی به فرم قطعی تبدیل شده است.

	a	b
S	A	-
A	E,I	F
E	E,I	F
F	E,I	F
I	-	T
(T)	-	-
E,I	E,I	F,T
(F,T)	E,I	F



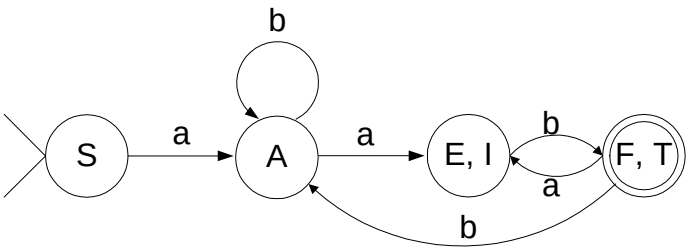
برای بهینه سازی ماشین خودکار فوق حالات مربوطه را به دو دسته پذیرشی و غیر پذیرشی به صورت زیر می‌توان تقسیم بندی نمود:

الف) 1- {FT} 2- {S, A, EI, F}

ب) 2- {FT} 2- {S} 3- {A, EI, F}

ج) 1- {FT} 2- {S} 3- {A, F} 4- {EI}

به این ترتیب مشاهده می‌کنید که حالات A و F با یکدیگر معادل هستند و یکی را می‌توان با دیگری جایگزین نمود. به این ترتیب ماشین خودکار بهینه برای عبارت به صورت زیر خواهد بود:



پرسش و پاسخ :

۲-۱ : کار تحلیلگر لغوی در یک کامپایلر را توضیح دهید ؟

۲-۲ : عبارات با قاعده چیست و به چه منظور به کار میرود ؟

۲-۳ : آیا تبدیل برنامه‌ها به زبان فارسی ممکن است ؟

۲-۱۰- تمرین

تمرین ۱- فرم کلی اعدادی را مشخص کنید که یا صفر می‌باشند و یا اینکه اگر طولشان بیش از یک است حتماً با یک رقم غیر صفر آغاز می‌شوند.

تمرین ۲ – فرم کلی اعداد در مبنای هشت را مشخص کنید در صورتی که بدانیم اینگونه اعداد حتماً با یک رقم صفر آغاز و سپس حرف O ظاهر می‌گردد و بعد از آن عدد مبنای هشت مشخص می‌شود. برای نمونه OO127 یک عدد مبنای ۸ است.

تمرین ۳- فرم کلی اعداد در مبنای شانزده در زبان C با رقم صفر و سپس X آغاز می‌شود. یک عبارت با قاعده برای بیان اعداد در مبنای شانزده ارایه نمایید. به طور مثال 0XAF25 یک عدد در مبنای ۱۶ است.

تمرین ۴- فرم کلی رشته‌ها را در زبان C با یک عبارت با قاعده معین کنید. توجه داشته باشید که در زبان C علامت (Backslash) \ در داخل یک رشته مفهوم خاص دارد.

تمرین ۵- فرم کلی اعداد صحیحی را تعیین کنید که در آنها ارقام به ترتیب صعودی ظاهر می‌شوند.

تمرین ۶- فرم کلی کلیهٔ رشته‌های اعداد مبنای دو را مشخص کنید که در آنها زیر رشته 011 ظاهر نگردد.

تمرین۷- کلیه رشته‌های اعداد مبنای دو را مشخص کنید که تعداد صفرها در آنها فرد و تعداد یک‌ها زوج باشد.

تمرین ۸- ماشین خودکار قطعی برای شناسه‌ها ترسیم نمایید.

تمرین ۹- ماشین خودکار قطعی بهینه برای عبارت (a | b | c)* abb ایجاد نمایید و سپس برنامه تحلیلگر لغوی را برای تشخیص اینگونه رشته‌ها بنویسید. ابتدا عبارت را به فرم غیرقطعی تبدیل نمایید و سپس قطعی و بهینه نمایید.

تمرین ۱۰- یک ماشین خودکار قطعی برای کلیه اعداد در مبنای دو که تعداد صفرهای آنها فرد و تعداد یک‌ها فرد است ایجاد کنید. در ضمن لااقل دو عدد صفر و دو عدد یک باید در این اعداد موجود باشد.

تمرین ۱۱- ماشین خودکار قطعی برای اعداد مبنای که حتماً دارای یک رقم صفر و یک رقم یک هستند را ایجاد نمایید به قسمی که اعداد با تعداد یک‌ها و صفرهای زوج را در حالت پذیرشی مشخص نماید. اعداد با تعداد یک‌های فرد و صفرهای زوج را در یک حالت پذیرشی مشخص نمایید. اعداد با تعداد یک‌های فرد و تعداد صفرهای فرد را نیز در حالت پذیرش دیگر ماشین مشخص کند. برای اعداد تعداد یک‌ها فرد و صفرها زوج و همچنین صفرها فرد و یک‌ها زوج نیز باید دو حالت پذیرش مجزا وجود داشته باشد. ابتدا برای هر یک از چهار حالت ذکر شده ماشینهای خودکار را ترسیم نمایید و سپس یک ماشین خودکار غیر قطعی برای ترکیب آنها ایجاد نمایید. سپس ماشین خودکار را قطعی کنید.

تمرین ۱۲- یک ماشین خودکار قطعی برای گرامر یکی از زبانهای C یا پاسکال ایجاد نمایید. سپس با استفاده از روش ارایه شده در این فصل یک تحلیلگر لغوی برای آن ایجاد کنید.

فصل سوم

زبان و گرامر

۳-۱- گرامر زبانها

زبانهای برنامه نویسی نیز همانند زبانهای محاوره‌ای مبتنی بر گرامر و قوانین خاص نحوی خود هستند. برای بیان قوانین گرامری زبانها از قوانین خاصی به نام Bacus Normal Form استفاده می‌شود، این فرم که در اصطلاح یک Meta Language نامیده می‌شود، اولین بار برای بیان قوانین گرامری زبان C استفاده شد. برای نمونه ساختار گرامری جملات را می‌توان به صورت زیر بیان کرد:

Program→Statement OtherSts

Statement→IfSt | WhileST | DoSt | ForSt | CompoundSt | AssignmentSt | CallSt

OtherSts→; Statement | λ

IfSt→IF Expression Statement ElsePart
ElsePart→else Statement | λ
WhileSt→while v Statement
DoSt→do Statement while v
ForSt→for(Statement; Expression ; Statement) Statement
CompoundSt→ '{' Statement '}'
AssignmentSt→id:= Expression
CallSt→id''()' | id'('ParamList')'
ParamList→id | id,ParamList
Expression→id | no

برای بیان قواعد گرامر فوق از چهار نوع علامت استفاده شده است:

۱- علامت → به مفهوم هست می‌باشد. برای نمونه قاعده شماره (۴) مشخص می‌کند که یک جمله While دارای قاعده گرامری به صورت ارائه شده است.

۲- علامت | به مفهوم یا است. برای نمونه (۳) مشخص می‌کند که قسمت ElsePart یا به صورت ELSE Statement است یا اصلاً وجود ندارد که با علامت λ نشان می‌دهند.

۳- علامت لامبدا λ به مفهوم تهی می‌باشد.

۴- علایم پرانتز (و) مهمولاً به عنوان جدا کننده به کار می‌روند. در گرامر فوق چون پرانتز بخشی از گرامر زبان است، لذا آن را در داخل کوتیشن قرار داده‌ایم.

گرامر فوق از تعدادی قاعده و یا در اصطلاح Production یا Rules تشکیل شده است. در سمت چپ هر قاعده یک ترم میانی قرار گرفته و سپس علامت هست یا → و سپس گسترش آن ترم میانی در سمت چپ فلش مشخص شده است. برای مثال به قاعده شماره (۶) توجه کنید. در این قاعده Statement سرترم گرامر است. اصولاً سه نوع ترم در داخل گرامر زبانها مشاهده می‌شود:

۱- ترم میانی (Non Terminal):

ترم میانی به آن دسته از گرامرها اطلاق می‌شود که بنا به گرامر زبان دارای تعریف و قاعده باشد و یا به عبارت دیگر در سمت چپ لااقل یک قاعده قرار بگیرد. به ترم میانی، ترم واسطه نیز گفته می‌شود، چرا که واسطه‌ای برای بیان و مشخص نمودن قوانین گرامری است. برای مثال در زبان فارسی جملهٔ اسنادی یک ترم میانی و در واقع واسطه‌ای برای بیان دستورالعمل‌های زبان فارسی است. در گرامر فوق ترم‌هایی مثل Expression، Cond و IfSt که در سمت چپ قواعد قرار گرفته‌اند، ترم‌های میانی هستند.

۲- ترم‌های پایانی (Terminal Symbols):
این دسته از ترم‌ها اصولاً در سمت چپ قواعد ظاهر نمی‌شوند و در واقع لغاتی هستند که در داخل زبان ظاهر می‌شوند. کار تشخیص آنها به عهدهٔ تحلیلگر لغوی می‌باشد. از این جمله می‌توان شناسه‌ها، اعداد، جملات تفسیری کامنت و لغات کلیدی زبان را نام برد. برای مثال در گرامر فوق ترم‌های پایانی از قبیل ELSE و FOR با حروف بزرگ مشخص شده‌اند.

۳- سرترم گرامر (Start Symbol):
سرترم یا ترم آغازین گرامرها، ترمی است میانی که شروع کنندهٔ یک گرامر است و نهایتاً در تعریف آن تمامی ترم‌ها به نحوی گنجانده شده‌اند. برای نمونه در گرامر فوق Statement سرترم گرامر است و IfSt سرترم نیست زیرا در تعریف Statement ترمی مانند IfSt یا به طور غیر مستقیم ترمی مانند RelOp وجود دارد اما در تعریف IfSt همواره کلیه جملات باید با کلمه IF آغاز شوند. اصولاً گرامرها یا مستقل از متن هستند یا وابسته به متن می‌باشند.

گرامر فوق مستقل از متن است. به این مفهوم که برای یک ترم میانی مهم نیست در چه متنی آمده و یا در کنار کدام ترم میانی ظاهر شود. یک ترم میانی مثل IfSt همیشه دارای ساختاری ثابت بوده و مستقل از متنی است که در آن ظاهر شده است. اما در گرامرهای وابسته به متن یا Context Sensetive ساختار یک ترم میانی ممکن است بسته به این که چه ترم‌هایی در اطراف آن قرار گرفته‌اند، فرق کند. برای مثال جملات If یا While مستقلاً ساختار خاص خود را دارند. اما اگر جمله If در کنار جمله While ظاهر شود، جمله حاصل ساختار دیگری خواهد داشت. به همین دلیل است که در گرامرهای وابسته به متن در سمت چپ قواعد ممکن است بیش از یک ترم میانی ظاهر شود.
در گرامرهای مستقل از متن همیشه یک ترم میانی در سمت چپ هر قاعده قرار می‌گیرد. مسلماً برای هر گسترش ترم میانی باید یک گسترش وجود داشته باشد. لذا در گرامر وابسته به متن اگر در سمت چپ قاعده، دو ترم میانی وجود داشته باشد در سمت راست باید تعداد ترم‌ها دو یا بیشتر باشد. اگر تعداد ترم‌ها در سمت چپ قاعده بتواند بیش از تعداد ترم‌ها در سمت راست باشد، آن گرامر را دیگر وابسته به متن نمی‌نامند. این نوع گرامر را نوع صفر می‌گویند. نوع دیگر گرامر باقاعده است. در این نوع گرامرها قواعد یا به صورت خود بازگشتی چپ و یا به صورت خود بازگشتی راست می‌توانند ظاهر شوند. البته قواعد غیر خود بازگشتی هم می‌تواند وجود داشته باشد. برای نمونه قاعده شماره (۶) در گرامر فوق خود بازگشتی راست است.

۳-۲- درخت‌های تجزیه
هدف از ایجاد درخت‌های تجزیه، تحلیل نحوی جملات است. به عبارت ساده‌تر با ایجاد درخت تجزیه صحت جملات از لحاظ قوانین گرامری مورد آزمون قرار می‌گیرد. این عمل با تجزیه جملات بر اساس عناصر آنها صورت می‌پذیرد. برای نمونه گرامر زیر را در نظر بگیرید:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow ID \mid NO. \mid (E)$$

گرامر فوق مبین ساختار کلی عبارات است، بنابراین هر یک از چهار عمل اصلی باید بر اساس این گرامر ایجاد شده باشند. برای نمونه عبارت زیر را باید بتوان بر اساس گرامر فوق ایجاد نمود.

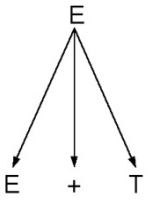
a*(b-c)+d/(e-f)

شاید روش تفکر یک معلم دستورالعمل زبان نیز به طریقی باشد که توضیح داده خواهد شد. یعنی اینکه جمله نوشته شده را تجزیه می‌کند و مشخص می‌نماید که آیا برای مثال یک جمله انگلیسی هست یا خیر. معمولاً ذهن انسان به دو روش عمل می‌نماید. یک

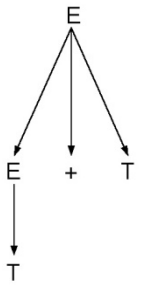
E

روش بالا به پایین است، یعنی اینکه با نگرش به جملهٔ داده شده و این نتیجه گیری که جمله باید یک عبارت باشد، عمل آغاز می‌شود. عمل تجزیه از سرترم گرامر یعنی E آغاز می‌شود. بنابراین در مرحله اول درخت تجزیه بصورت زیر است:

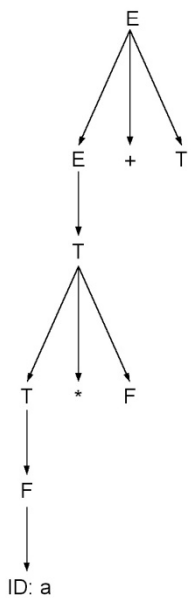
حالا با در نظر گرفتن اینکه عبارت داده شده حاصل جمع دو ترم d/(e-f) و a*(b-c) است، سر ترم E بنابر قاعدهٔ E→E+T به صورت زیر گسترش داده می‌شود:



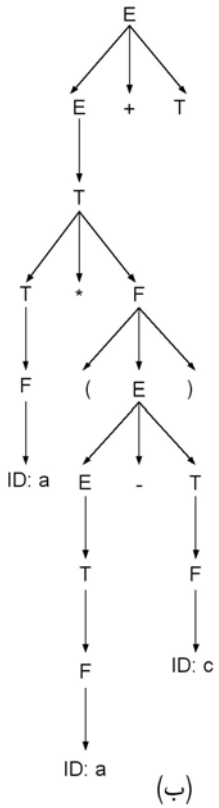
همانگونه که در بالا توضیح داده شد، عبارت داده شده حاصل جمع دو ترم است لذا بنابر قاعده E→T ترم میانی E به T گسترش داده می‌شود.



اکنون باید بتوان از ترم میانی T عبارت a*(b-c) را تولید کرد لذا، درخت تجزیه فوق بصورت (الف) توسعه داده می‌شود.
اکنون باید بتوان F را با (b-c) جایگزین کرد. بنابراین درخت تجزیه فوق بصورت (ب) گسترش داده می‌شود.

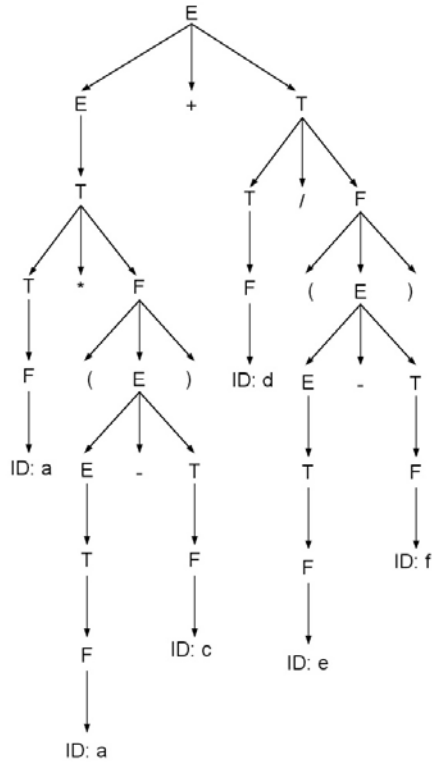


(الف)



(ب)

اکنون با توجه به اینکه پس از علامت جمع باید از ترم میانی T نهایتاً عبارت d/(e-f) حاصل شود درخت تجزیه به صورت (ج) توسعه داده می‌شود.



(ج)

همانگونه که در شکلهای فوق مشاهده می‌شود با استفاده از روش بالا به پایین و تجزیه یک عبارت بر طبق گرامر نهایتاً درخت تجزیه که در رأس آن سرترم گرامر و در برگها ترم‌های پایانی قرار دارند، ایجاد شده است.

چنانچه عبارت از لحاظ گرامری غلط می‌بود، این امکان وجود نداشت که بتوان بر اساس گرامر درخت فوق را ایجاد کرد. مراحل تجزیه با ایجاد مرحله به مرحله درخت نمایش داده شد. زیرا با توجه به درخت تجزیه نهایی نمی‌توان مراحل تجزیه را مشخص نمود. در طی مراحل ایجاد درخت تجزیه، عمل تجزیه ترم‌های میانی از چپ به راست انجام شده، همواره سمت چپ‌ترین عنصر یا گره در داخل درخت گسترش داده شد، تا نهایتاً بتوان به یک ترم پایانی بعدی در داخل جمله داده شده از سر ترم گرامر مشتق یا نتیجه گیری شده است. مراحل تجزیه سر ترم تا رسیدن به ترم‌های پایانی درون جمله داده شده را در اصطلاح مراحل اشتقاق یا Derivation می‌گویند. مراحل اشتقاق سر ترم E تا رسیدن به جمله فوق به صورت زیر است:

$$E \rightarrow E + T \rightarrow T + T \rightarrow T * F + T \rightarrow F * F + T \rightarrow a * F + T \rightarrow a * (E) + T \rightarrow A * (E - T) + T \rightarrow a * (T - T) + T \rightarrow a * (T - T) + T \rightarrow a* (F - T) + T \rightarrow a* (b - T) + T \rightarrow A * (b - F) + T \rightarrow a * (b - c) + T \rightarrow a * (b - c) + T / F \rightarrow a * (b - c) + F / F \rightarrow A * (b - c) + d / F \rightarrow a * (b - c) + d / (E) \rightarrow a * (b - c) + d / (E - T) \rightarrow A * (b - c) + d / (T - T) \rightarrow a * (b - c) + d / (F - T) \rightarrow a * (b - c) + d / (e - T) \rightarrow A * (b - c) + d / (e - F) \rightarrow a * (b - c) + d / (e - f)$$

همانگونه که مشاهده نمودید، عمل اشتقاق از سرترم آغاز و نهایتاً به جمله داده شده خاتمه می‌یابد. در این میان فرمهای جمله‌ای که حاوی ترمهای میانی و پایانی و یا فقط ترم‌های میانی هستند، ایجاد می‌گردد. فرمهای جمله‌ای را Sentential form نیز می‌گویند.

۳-۴- تجزیه پایین به بالا

در روش تجزیه پایین به بالا برخلاف روش پایین به بالا، عمل تجزیه از پایین و از ترم‌های درون جمله آغاز و بـه سـرترم گرامـر خاتمـه می‌یابد. لذا این روش پایین به بالا است. برای نمونه عبارت زیر را در نظر میگیریم:

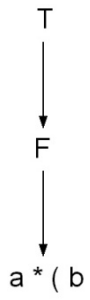
a*(b-c)+d

هدف تجزیه این عبارت به روش پایین به بالا است. همانگونه که توضیح داده شد در روش پایین به بالا عمل تجزیه از چپ به راست بـا جایگزینی ترم‌های پایانی با میانی بر اساس سمت چپ قواعد انجام می‌شود و آنقدر عمل جایگزینی ادامه می‌یابد تا در صـورت صـحت، بتوان به سرترم داده شده رسید.

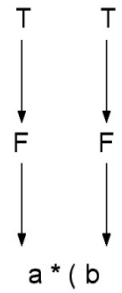
ابتدا سمت چپ‌ترین لغت از عبارت داده شده یعنی a توسط تحلیلگر لغوی خوانده می‌شود. بر سر ورودی علامت * ظاهر می‌شود. طبق قاعده T → T * F، قبل از عملکرد * فقط یک T می‌تواند ظاهر شود . لذا ، با استفاده از قاعده F→ id ترم پایانی a که یک شناسه یا id است را با F و سپس طبق قاعده T→ F ترم میانی F را با Tمی‌توان جایگزین نمود. اکنون ترم بعـدی یعنـی * از سـر ورودی خوانـده می‌شود. طبق قاعده T→T*F پس از * ترم میانی F می‌تواند ظاهر شود. لذا، ترم بعدی بایستی F باشد.

اکنون ورودی '(b-c)+d' است. بر سر ورودی لغت) وجود دارد. پس از خواندن) و b درخت بصورت (الف) خواهد بود.

حال بر سر ورودی لغت منها وجود دارد. طبق قاعده E→E-T قبل از عملگر – باید یک عبارت E وجود داشته باشد. لذا ترم پایانی b که یک شناسه یا id است. بنابر قاعده F → id با ترم میانی F و سپس طبق قواعـدT → F و E → T نهایتـاً بـا تـرم میـانی E جـایگزین می‌شود. بنابراین، تا این مرحله درخت تجزیه پایین به بالا به صورت (ب) خواهد بود.

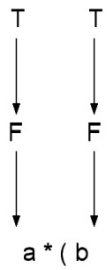


(الف)

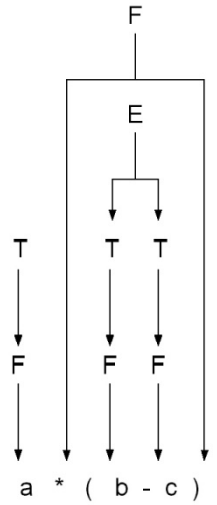


(ب)

اکنون ورودی d + (c – است. عملگرهای – و سپس c از ورودی خوانده می‌شود. طبق قاعده E→ E-T باید پس از – در ورودی یک ترم ظاهر شود. لذا، c با F وF با T و نهایتاً T – E با E جایگزین خواهند شد. حال (از ورودی خوانده می‌شود و به این ترتیب در



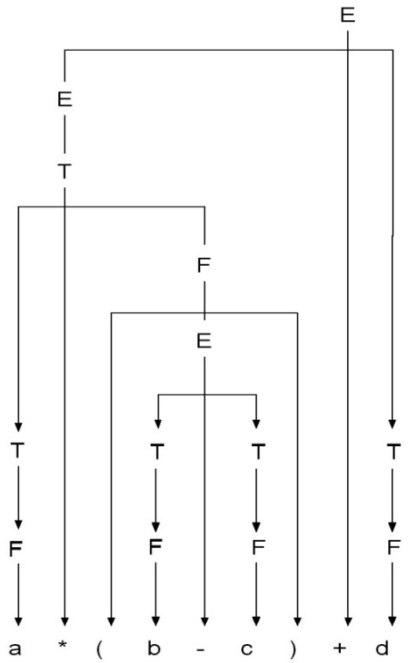
داخل درخت تجزیه (E) ظاهر می‌شود و طبق قاعده F → (E) می‌توان آن را با F جایگزین کرد.



اکنون ورودی '-c)+d' است. عملگرهای – و سپس c از ورودی خوانده می‌شوند. طبق قاعده E → E-Tباید پس از - در ورودی یک ترم ظاهر شود. لذا c با F و F با T و نهایتاً E-T با E جایگزین خواهند شد.

حال (از ورودی خوانده می‌شود و به این ترتیب در داخل درخت تجزیه (E) ظاهر می‌شود و طبق قاعدهٔ (E) → F می‌توان آن را با F جایگزین کرد.

اکنون ورودی 'd+' است. بر سر ورودی علامت + وجود دارد. قبل از + طبق قاعده E→ E + T باید یک E وجود داشته باشد. بنابراین T * F را که اکنون در رأس درخت تجزیه وجود دارد طبق قاعده T→ T * F با T و سپس طبق قاعده E → T ترم T را با E می‌توان جایگزین نمود. نهایتاً درخت تجزیه به صورت زیر تبدیل می‌شود:



همانگونه که مشاهده نمودید عمل تجزیه از ترمهای پایانی آغاز و به ترم گرامر خاتمه یافت. این گونه تجزیه را در اصطلاح تجزیه پایین به بالا یا Bottom Up Parsing گویند.

طبق تعریف، مراحل اشتقاق نمایانگر مراحل رسیدن از سرترم گرامر به ترم‌های پایانی درون جمله مورد نظراست، در تجزیه پایین به بالا، مراحل اشتقاق درست برخلاف مراحل تجزیه است چرا که در اینجا نهایتاً به سر ترم گرامر می‌رسند در صورتی که مراحل اشتقاق از سر ترم گرامر آغاز می‌شود. به عبارت دیگر مراحل اشتقاق نشان می‌دهد که در طی چه مراحلی جمله داده شده از سر ترم گرامر مشتق شده است. مراحل مشتق کردن a*(b-c)+d از سرترم به صورت زیراست. باید توجه داشته باشید که مراحل اشتقاق درست برخلاف مراحل تجزیه است. بنابراین اگر از آخرین مرحله ایجاد درخت تجزیه به مراحل قبلی برگشت شود مراحل اشتقاق بصورت زیرمعین می‌شود :

$$E \rightarrow E+T \rightarrow E+F \rightarrow E+d \rightarrow T+d \rightarrow T*F+d \rightarrow T*(E)+d \rightarrow T*(E-T)+d \rightarrow T*(E-F)+d \rightarrow T*(E-c)+d \rightarrow$$
$$T*(E-c)+d \rightarrow T*(T-c)+d \rightarrow T*(F-c)+d \rightarrow T*(b-c)+d \rightarrow F*(b-c)+d \rightarrow a*(b-c)+d$$

همانگونه که مشاهده می‌شود در طی مراحل اشتقاق برخلاف اشتقاق چپ که قبلاً توضیح داده شد، همواره سمت راست‌ترین ترم‌ها تا رسیدن به ترم پایانی درون جمله داده شده جایگزین می‌گردند. لذا این گونه اشتقاق را اشتقاق راست گویند. اولین ترم پایانی که در طی این مراحل اشتقاق درفرمهای جمله‌ای ظاهر شد ترم پایانی d در فرم جمله‌ای T+d بود و سپس از راست به چپ ترم پایانی بعدی یعنی c درفرم جمله ای T* (E –c) +d ظاهرگردید لذا همانگونه که مشاهده می‌کنید جایگزینی ازسمت راست به چپ انجام شده است.

۵-۳- گرامرهای مبهم

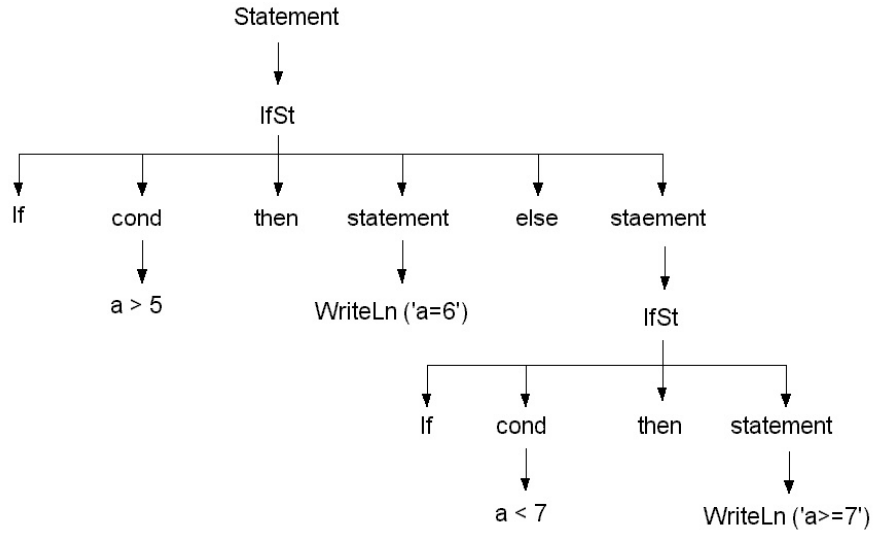
چنانچه بتوان برای جمله داده شده براساس گرامر زبان، بیش ازیک درخت تجزیه ایجاد نمود، آن را مبهم می‌نامند. در حالت کلی با نگرش به گرامرها نمی‌توان ابهام را تشخیص داد. اما باید دید که چرا. اگربتوان بیش ازیک درخت تجزیه برای جمله داده شده ایجاد نمود، گرامر ابهام دارد. این باید یک مزیت باشد، چرا آن را ابهام می‌خوانند؟ به عنوان یک مثال ساده به گرامر زبانهای C یا پاسکال برای جملات شرطی if توجه نماید:

Statement → IfSt | WhileSt | CompoundSt | AssignmentSt | CallSt
IfSt → IF (Cond) statement | IF (Cond) Statement else Statement
Cond → Exp | Exp Relop Exp

برای نمونه به جمله زیر توجه نمایید:

```
if (a>5)if (a<7) cout<<"a=6"; else cout<<"a>=7"
```

بنابراین گرامر فوق درخت تجزیه بصورت زیر می‌تواند باشد:

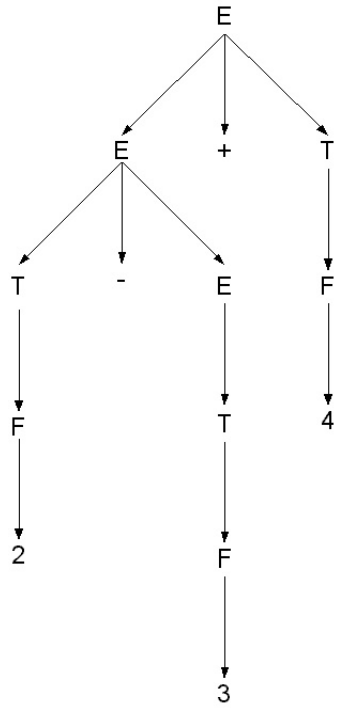


درخت تجزیه را به صورت زیر نیز می‌توان تولید کرد:

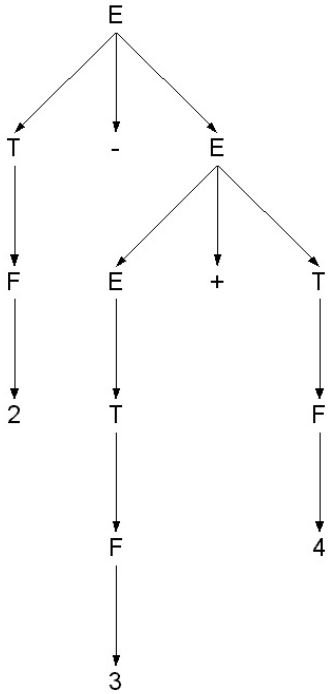
می‌خواهیم برای جملهٔ 4+3-2 درخت تجزیه ترسیم نماییم:

همانگونه که مشاهده می‌کنید ایجاد دو نتیجه متفاوت ۵- و ۳ برای عبارات فوق به خاطر مبهم بودن گرامر و در واقع قاعده بـه صـورت خودبازگشتی چپ و خودبازگشتی راست برای ترم میانی E است.

بنابرگرامر If دو درخت تجزیه با دو مفهوم متفاوت ایجاد شده است. در درخت تجزیه شکل اول، Else وابسته به If خـارجی مـی‌باشـد و



درخت با حاصل (2-3)+4 = 3



درخت با حاصل 2-(3+4) = -5

درشکل دوم، Else وابسته بهIf داخلی است. برای رفع این مشکل درزبان‌های C و پاسکال، برای تکمیل گرامر بطور ضمنی و نه بر طبق گرامر، تعیین کرده‌اند که Else به نزدیکترین If وابسته است. بنابراین درخت تجزیه شکل دوم توسط کامپایلر پاسکال ایجاد می‌شود. درزبان فورترن ۷۷ با افزایش Endif این مشکل حل شده و گرامر به صورت زیر تبدیل گردیده است:

If → IF Cond THEN Statement ENDIF | IF Cond THEN Statement ELSE Statement ENDIF

بنابراین اگر قرار است که در جمله یادشده Else وابسته به If خارجی باشد، به صورت زیر عمل می‌شود:

```
If a>5 THEN If a<7 THEN WriteLn('a=6') ENDIF ELSE WriteLn('a>=7') ENDIF
```

بالعکس اگر Else وابسته به If داخلی باشد، جمله به صورت زیر نوشته می‌شود:

```
If a>5 THEN If a<7 THEN WriteLn('a=6') ELSE WriteLn('a>=7') ENDIF
```

اصولاً در حالت کلی، ابهام در گرامرها را نمی‌توان نشان داد و اثبات نمود، اما نکته قابل توجه این است که اگر گرامری به صورت خودبازگشتی چپ و خودبازگشتی راست برای یک ترم میانی قواعدی داشته باشد، آن گرامر مبهم است.

I) A → Aa | bA

همچنین اگر در یک گرامر نهایتاً بتوان طی مراحل استنتاج یا اشتقاق از یک ترم میانی به خود آن ترم رسید، گرامر مبهم می‌باشد.

2) A → a → . . . . → A

اگر در گرامری قواعد به صورت خودبازگشتی راست یا چپ برای یک ترم میانی وجود داشته باشد و علاوه بر آن قاعده دیگر برای این ترم میانی به صورتی باشد که حالت خودبازگشتی در دو سطح دو ترم در سمت راست آن قاعده برای آن ترم میانی وجود داشته باشد و در اصطلاح قاعده به صورت Self Embeding یا خودگردان باشد، باز هم گرامر مبهم خواهد بود.

3) A → Aa | bAd

برای نمونه به گرامر مبهم عبارات توجه کنید:

E → E + T | T – E | T
T → T * F | T / F | F
F → ID | NO. | (E)

پرسش و پاسخ :

۳-۱ : ترم میانی چیست ؟
۳-۲ : ترمهای پایانی را تعریف کنید ؟
۳-۳ : منظور از سر ترم گرامر چیست ؟
۳-۴ : چه گرامری را مبهم گویند ؟

۳-۶ تمرین

تمرین ۱- مراحل استنتاج را برای عبارت (a*b-c)/(d-e) در روش‌های تجزیه بالا به پایین و پایین به بالا مشخص نمایید.

تمرین ۲- گرامر ساختار لیست را در نظر بگیرید،

S → '(' L ')' | a
L → L ',' S | S

درخت تجزیه را برای جملات زیر ترسیم نمایید:

الف – (a , a)
ب - (a , (a , a))
ج - (a , ((a , a) , (a , a)))

تمرین ۳- نشان دهید که گرامر زیر مبهم است.

S → aSbS → bSaS | l

تمرین ۴- گرامر زیر را در نظر بگیرید :

bexpr → bexpr or bterm → bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false

الف ـ درخت تجزیه برای عبارت (not (true or false) ایجاد کنید.
ب ـ آیا این گرامر مبهم است؟

تمرین ۵ ـ چرا اگر در یک گرامر برای یک ترم میانی قواعد به دو صورت خود باز گشتی چپ و خود بازگشتی راست وجود داشته باشد آن گرامر مبهم است.

فصل چهارم

تجزیه بالا به پایین

۴-۱- تحلیلگر ذهن

عملکرد تجزیه بالا به پایین بر مبنای روش تفکر مغز و ذهن آدمی برای تحلیل نحوی جملات استوار است. فردی فارسی زبان در مقابل شما قرار می‌گیرد . می‌خواهد با شما صحبت کند به دهان وی می‌نگرید. پیش بینی می‌کنید که جمله فارسی می خواهد بیان کند. جمله فارسی سر ترم جملات در دستور العمل زبان فارسی است زیرا ، هر گفته ای در زبان فارسی باید جمله فارسی باشد. بنابراین مغز پیش بینی می‌کند که جمله خارج شده از دهان سر ترم گرامر جملات در زبان فارسی باید باشد.

حالا فرض کنید که فرد ' اگر' از دهانش خارج شود. بلافاصله تحلیلگر نحوی مغز به سراغ مجموعه لغات یا در اصطلاح ترمهای پایانی ای می‌رود که می توانند جملات فارسی را آغاز کنند. کلمه ' اگر' می تواند آغاز کننده یک جمله فارسی باشد . در اصطلاح مجموعه ترم های پایانی که می‌توانند آغاز کننده یک ترم باشند را مجموعه سرآغاز یا مجموعه First برای آن ترم گویند.

پس از اطمینان از اینکه لغت 'اگر' درمجموعه سر آغاز یا First جمله فارسی است باید مشخص نمود که کدام گسترش از گسترشهای متفاوت جمله فارسی با کلمه اگر آغاز می‌شود. پاسخ جملات شرطی است. کلمه اگر متعلق به مجموعه سرآغاز یا First جملات شرطی است. حالا طبق گرامر جملات شرطی پس از کلمه ' اگر' انتظار شنیدن یک شرط می‌رود. باز هم بر طبق گذشته تحلیلگر نحوی ذهن کلمه بعدی را گرفته در مجموعه سرآغاز شرط آنرا جستجو می‌کند. و به این ترتیب مراحل ادامه می یابد.

۴-۲- ایجاد الگوریتم تحلیل نحوی بر مبنای عملکرد ذهن

بطور خلاصه در بخش قبل ملاحظه نمودید که برای انجام عمل تحلیل نحوی مغز به صورت زیر عمل می‌نماید:

۱. انتظار سر ترم گرامر زبان را در آغاز دارد. سر ترم گرامر زبان فارسی جمله فارسی است.
۲. لغت اولی دریافت می‌شود. در بخش قبل در حالی که انتظار جمله فارسی را تحلیلگر ذهن می‌داشت لغت ' اگر ' دریافت شد.
۳. در داخل مجموعه First برای ترم مورد انتظار بدنبال لغت دریافت شده می گردد.
۴. در صورت یافتن لغت، در داخل مجموعه First برای گسترش های متفاوت ترم میانی (در مثال فوق جمله شرطی با لغت ' اگر' آغاز می‌شد).
۵. حالا با مشاهده ترم پایانی دریافت شده در گرامر، طبق گرامر ترم مورد پیش بینی را مشخص می‌کند. در مثال فوق با یافتن کلمه ' اگر' طبق گرامر انتظار مشاهده یک شرط می‌رفت. لذا، لغت بعدی دریافت شده و در صورت عدم خاتمه جمله از مرحله ۲ عملیات تکرار می‌شود.برای نمونه به گرامر زیر توجه کنید.:

Program → Statement OtherSts

Statement→IfSt | WhileST | DoSt | ForSt | CompoundSt | AssignmentSt | CallSt

OtherSts → ; Statement | λ

IfSt → IF Expression Statement ElsePart
ElsePart → else Statement | λ
WhileSt → while v Statement
DoSt → do Statement while v
ForSt → for(Statement; Expression ; Statement) Statement
CompoundSt → '{' Statement '}'
AssignmentSt → id:= Expression
CallSt → id'('')' | id'('ParamList')'
ParamList → id | id,ParamList
Expression → id | no

برای تجزیه جمله

If (A) C = 1

بر اساس عملکرد ذهن به صورت زیر عمل می‌شود.

- بر اساس گرامر در ابتدا انتظار مشاهده سر ترم گرامر یعنیProgram داریم که با Statement شروع می‌شود.
- اولین لغت یعنی if از تحلیلگر لغوی دریافت می‌شود.
- تحلیلگر نحوی به جستجوی لغت if داخل مجموعه لغات آغاز کننده ترم مورد انتظار یعنی می پردازد

First(statement) = First(IfSt) + First(WhileSt) + First(DoSt) +
First(ForSt) + First(CompoundSt) + First(AssignmentSt) + First(CallSt)
= [S_If,S_While, S_Do, S_For, S_BEGIN, S_Id] ;

- با مشاهده نوع لغت If در مجموعه First(statement) تحلیلگر لغوی بدرون مجموعه سراغاز برای گسترشهای متفاوت statement می‌نگرد و لغت if را در جمله (First(IfSt پیدا می‌کند.بنابر این تحلیلگر تشخیص می‌دهد که جمله مورد نظر یک جمله شرطی If است. درخت زیر بر اساس گسترش موجود برای جمله IfSt بر طبق گرامر، ایجاد میشود:

- حالا پس از مشاهده if، بنابر گرامر جمله شرطی ifSt در ورودی انتظار مشاهده Expression می‌رود. از تحلیلگر لغوی لغت بعدی یعنی A دریافت می‌شود. درون مجموعه سر آغاز عبارات یعنی:

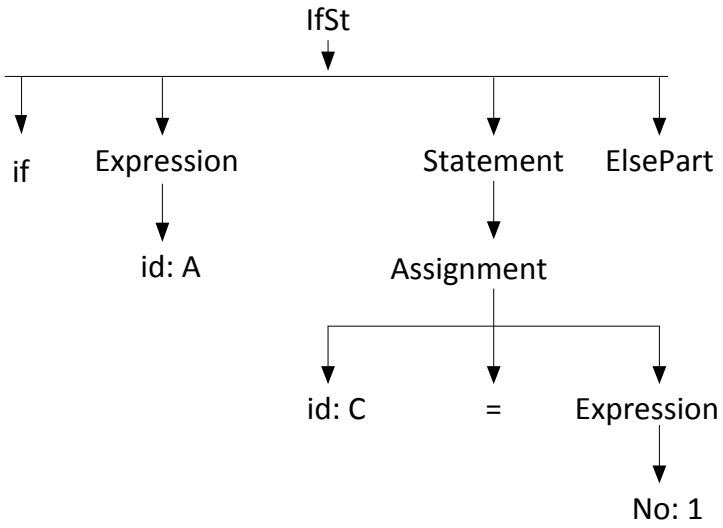
IfSt

First(Expression)= { S_Id , S_No}

نوع لغت A یعنی S_Id وجود دارد. بنابر این تحلیلگر نحوی به کار خود ادامه می‌دهد.

If Expression statement ElsePart

حالا تحلیلگر پیش بینی مشاهده یک Statement را بر اساس گرامر می‌نماید. تحلیلگر نحوی از تحلیلگر لغوی لغت بعدی را دریافت می‌کند. لغت C از نوع S_Id را تحلیلگر نحوی دریافت می‌کند. این لغت در مجموعه First(statement) وجود دارد اما، نکته اینجا است که هر دو گسترش CallSt و Assignment با Id آغاز می‌شوند. لذا، تحلیلگر نحوی نمی‌تواند بر اساس ترم پیش بینی id تصمیم بگیرد که کدام گسترش باید انتخاب شود. بنابر این ترم بعدی را دریافت می‌کند ترم بعدی در مثال فوق '=' است. حالا می‌توان تصمیم گرفت که گسترش AssignmentSt باید انتخاب شود. نهایتاً درخت تجزیه بصورت زیرخواهد بود.



۴-۳- نتیجه تحلیل عملکرد ذهن

از مطالب ارائه شده در بخش قبلی می‌توان سه نتیجه به شرح زیر گرفت:

اولاً در هر مرحله تحلیلگر پیش بینی می‌کند که ترم بعدی چه باید باشد. برای نمونه تحلیلگر نحوی در ابتدا پیش بینی سر ترم گرامر را می‌نمود. به این نوع تحلیلگر ها در اصطلاح تجزیه گرهای پیش‌بینی‌کننده یا Predictive Parsers گویند.

ثانیاً عمل خواندن لغات از چپ به راست صورت می‌گیرد. در مثال فوق ابتدا سمت چپ ترین لغت یعنی if از ورودی خوانده شد و عمل خواندن از چپ به راست ادامه پیدا کرد. لغت خوانده شده از ورودی را در اصطلاح ترم پیش بینی یا LookAhead گویند زیرا، بر اساس این لغت است که تحلیلگر پیش بینی میکند ترم بعدی چه باید باشد. برای نمونه بر اساس ترم پیش‌بینی if تحلیلگر نحوی تصمیم گرفت که گسترش IfSt را برای Statement باید انتخاب نماید.پس پیش‌بینی مشاهده جمله ifSt در ورودی شد.

ثالثا با در دست داشتن یک یا بیشتر از ترم های پیش بینی تحلیلگر می‌تواند تصمیم گیری کند که کدام گسترش از گسترشهای متفاوت ترم مورد انتظار را باید انتخاب نماید. برای نمونه در بالا با در دست داشتن یک ترم پیش بینی If تحلیلگر تصمیم گرفت که گسترش را برای ترم مورد پیش بینی یعنی Statement باید انتخاب کند اما، در هنگامیکه در ورودی ترم پیش بینی از نوع 'id' ظاهر شد چون دو گسترش متفاوت Statement یعنی AssignmentSt و CallSt هر دو با ترم 'id' آغاز می‌شدند، تحلیلگر نمی‌توانست تصمیم بگیرد که کدامیک را انتخاب کند. در این حالت، با در دست داشتن دو ترم پیش بینی' id 'و ' =' تحلیلگر توانست تصمیم بگیرد که کدام گسترش باید انتخاب شود .

اصولاً عمل تجزیه بالا به پایین از سر ترم گرامر آغاز و به ترم های پایانی درون جمله یا برنامه مورد کامپایل خاتمه می یابد. برای انجام عمل تجزیه بالا به پایین فرم کلی جملات یا به عبارت دیگر گرامر زبان را آنقدر محدود می کنند که، با در دست داشتن k ترم پیش بینی از متن برنامه مورد کامپایل بتوان عمل تحلیل نحوی را بدرستی انجام داد.

تجزیه گرهای پیش‌بینی‌کننده یا Perdicitive Parsers با در دست داشتن یک یا چند ترم پایانی بعدی در ورودی، قادر به انجام عمل تحلیل نحوی هستند. گرامرهای مورد استفاده برای این روش تجزیه را اصطلاحاً LL(K) یا Left Lookahead گویند. منظور این است که می‌توان با استفاده از قواعد گرامر و با در دست داشتن k ترم پایانی بعدی در ورودی یا در اصطلاح k ترم پیش بینی، عمل تجزیه بالا به پایین را انجام داد.

۴-۴- گرامرهای LL(1)

تجزیه گرهای پیش‌بینی‌کننده برای تجزیهٔ بالا به پایین با در دست داشتن یک یا چند ترم بعدی در ورودی باید قادر به

تشخیص این باشند که:

اولاً: آیاجملهٔ مورد نظر تا این مرحله از لحاظ گرامری صحیح است.

ثانیاً: چه ترمها ی میانی مورد انتظار هستند.

ثالثاً : چه ترمهای پایانی در سر ورودی می‌توانند ظاهر شوند.

چنانچه با در دست داشتن حداکثر k ترم بعدی از ورودی یا بعبارت دیگر با در دست داشتن حداکثر k ترم پیش‌بینی بتوان عمل تجزیه قابل پیش بینی ای را بر روی یک گرامر به انجام رساند،آن گرامر را LL(k)گویند.

چنانچه با در دست داشتن یک ترم پایانی بعدی از چپ به راست در ورودی بتوان تشخیص داد که از بین گسترش های متفاوت برای یک ترم میانی کدامیک باید انتخاب شوند، گرامر را LL(1) گویند.

حال برای نمونه گرامر روبرو را در نظر بگیرید:

S ⟶ I | W | A | P | C
I ⟶ if (B) S E
B ⟶ id D
D ⟶ R id | λ
R ⟶ < | > | ==
W ⟶ While (B) S
A ⟶ id = No
P ⟶ id '(' M ')'
M ⟶ id | λ
C ⟶ '{' 'T' '}'
T ⟶ S G
G ⟶ ; T | λ
E ⟶ ;else S | λ

این گرامر LL(1) نیست، زیرا برای نمونه با دیدن id یا شناسه نمی توان مشخص نمود که از بین قواعد مختلف برای گسترش S کدامیک باید انتخاب شود. اصولاً جهت تجزیهٔ بالا به پایین باید پس از خواندن یک ترم پایانی از ورودی مشخص نمود که آیا ترم پایانی خوانده شده متعلق به مجموعهٔ First برای ترم پایانی یا میانی مورد انتظار است یا خیر ؟ سپس باید مشخص نمود که کدام گسترش از گسترش های متفاوت ترم میانی مورد انتظار، با ترم پایانی خوانده شده، آغاز می شوند. به عبارت دیگر باید مشخص نمود که ترم پایانی خوانده شده، متعلق به مجموعهٔ First برای کدامیک از گسترش های مختلف ترم میانی مورد نظر است.

بنابراین اگر قرار باشد با داشتن یک ترم پیش بینی در هر مرحله از تجزیه بتوان مشخص نمود که کدام گسترش برای ترم میانی A با گسترش های

$A \rightarrow A1 | A2 | A3 | \ldots | An$

باید انتخاب شود. آنگاه باید اشتراک مجموعهٔ First برای هر دو گسترش متفاوت از A تهی باشد یا بعبارت دیگر باید رابطهٔ:

$$\forall i, j \varepsilon 1..n, i \neq j \Rightarrow first(Ai) \cap first(Aj) = \phi$$

برقرار باشد. در غیر اینصورت فرض کنید که برای نمونه

$$i, j \varepsilon 1..n, \neq j, first(Ai) \cap first(Aj) = \{a\}$$

آنگاه با مشاهدهٔ ترم پایانی a در ورودی تحلیلگر نمی‌تواند بلافاصله تصمیم بگیرد که آیا گسترش A به Ai باید انتخاب شود، یا گسترش A به Aj .

پس بطور خلاصه می‌توان گفت :

یک گرامر LL(1) است اگر اشتراک مجموعه First هردو گسترش متفاوت برای هر ترم میانی متعلق به آن گرامر تهی باشد.

برای نمونه درخت تجزیه را برای جملهٔ:

{ a = 5 ; if (b) f( ) }

با روش بالا به پایین می‌توان بصورت زیر ایجاد نمود:

۱- تحلیلگر لغوی فراخوانی می‌شود.

۲- لغت '{' توسط تحلیلگر لغوی به تحلیلگر نحوی برگردانده می‌شود.

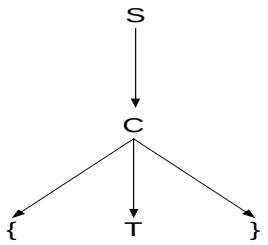
۳- تحلیلگر نحوی که در انتظار مشاهده S در ورودی است ، ابتدا می‌بایست مطمئن شود کـه تـرم خوانـده شـده یعنی '{' آیا متعلق به مجموعهٔ First (S) است.

'{' ε first (S) = { if , while , id , '{' }

سپس باید مشخص شود که '{' به مجموعهٔ First برای کدام گسترش از گسترش های متفاوت S تعلق دارد. چون:

'{' ε first (C) = { '{' }

پس درخت تجزیهٔ زیر ایجاد می‌شود :

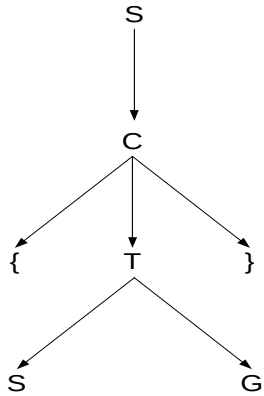


۴-با مشاهدهٔ { در ورودی دو مرتبه مراحل ۱ تا ۴ بشرح زیر تکرار می‌شود.

- لغت بعدی a می باشد که از نوع id است.

$$id \in first(T) = first(S) = \{if, while, id, '\{'\}$$

بنابراین گسترش T به S G انتخاب شده درخت تجزیه بصورت زیر خواهد بود.



اما در این مرحله مشاهده می‌شود که دو گسترش مختلف S → A و S → P هردو با یک ترم id آغاز می شوند، بنا بر ایـن دراین مرحله به علت L L(1) نبودن گرامر نمی‌توان با در دست داشتن ترم پیش‌بینی a تصمیم قطعی در مورد گسترش S گرفت.

همانگونه که در بالا توضیح داده شد، برای انجام تجزیهٔ بالا به پایین نیاز به در دست داشتن مجموعهٔ First برای ترمهای گرامر است. روش محاسبهٔ مجموعهٔ سرآغاز یا مجموعهٔ First در بخش بعدی توضیح داده خواهدشد.

۴-۵- مجموعه های سرآغاز و پیرو

مجموعهٔ سرآغاز برای یک ترم شامل مجموعه ترم های پایانی است که گسترش های متفاوت و جملات مشتق از آن ترم را مـی تواننـد آغازکنند. در حالت کلی برای ترم A مجموعهٔسرآغاز بصورت زیر محاسبه می‌شود:

- چنانچه برای A گسترشی بصورت A → a α وجود داشته باشد، آنگاه :

First (A) = { a }

- چنانچه برای A گسترشی بصورت A → B α وجود داشته باشد ، آنگاه :

First (A) = First(B)

- چنانچه برای A گسترشی بصورت A → B α وجود داشته باشد و B → λ | β ، آنگاه :

واضح است که اگر B تهی یا λ در نظر گرفته شود، آنگاه قاعده A → B α در واقع بصورت A → α تبـدیل مـی‌شـود. بنـابراین First(A) شامل First(α) هم می‌شود.

First (A) = First(β) +First(α)

- چنانچه برای A گسترشی بصورت A → X1 X2 X3 ... Xn وجود داشته باشد و برای X1 تا Xi قاعده تهی هم وجوداشته باشد یا به عبارت دیگر

$$\forall 1 \le j \le i \quad X_j \to \beta_j \mid \lambda$$

آنگاه:

First (A) = First(X1) +First (X2) + ... +First (Xi+1)

واضح است که در قاعده A → X1 X2 X3 ... Xn اگر X1 وجود داشته باشد،آنگاه First (A) = First (X1)

وگرنه در صورتی که X1 تهی یا λ باشد،آنگاه قاعده بصورت زیر خواهد بود :

A → X2 X3 ... Xn

به این ترتیب:

First (A) = First (X1) +First (X2)

حالا اگر X1 وX2 هردو تهی باشند آنگاه

First (A) = First (X1) +First (X2) +First (X3)

به همین ترتیب می‌توان نهایتاً نشان داد که :

First (A) = Σfirst (Xj) , 1≤ j ≤ i+1

چنانچه برای ترم A گسترش تهی نیز وجود داشته باشد یا به عبارت دیگر در حالت کلی قواعد مربوط به ترم A به صورت A → α | λ باشد، آنگاه با مشاهدهٔ یک عنصر متعلق به First (α) واضح است که گسترش A → α باید انتخاب شود. امـا چـه عنصری باید ظاهر شود تا در حالیکه انتظار مشاهده A می رود، بتوان دریافت که گسترش A → λ باید انتخـاب شود.واضـح است که اگر ترم مورد انتظار یعنی A در ورودی ظاهر نشود باید ترمی که پس از A انتظـار مشـاهده آن در ورودی مـی رفـت ظاهر شود. برای نمونه به گرامر زیر توجه کنید :

LabeldSt	⟶	Label Statement
Label	⟶	id: | λ
Statement	⟶	AssignmentSt | IfSt | WhileSt
AssignmentSt	⟶	id = Expression
WhileSt	⟶	WHILE (Expression) Statement
ifSt	⟶	IF (Expression) Statement

طبق گرامر فوق جملات دارای برچسب یا Label و بدون برچسب هستند. در آغاز کار طبق گرامر فوق جهت مشاهده LabelSt ابتدا انتظار مشاهده یک Label در ورودی می‌رود. بنابراین انتظار می‌رود که اولین ترم در ورودی یک id باشد زیرا :

id ε First(Label)={id, λ }

اما، اگر ترم خوانده شده از ورودی از نوع id نباشد، مشخص است که Label در ورودی وجود نداشته است. بنابر ایـن انتظـار می رود که ترم خوانده شده آغاز کنندهٔ ترم بعد از Label یعنی Statement باشد. بنابراین چنانچه ترم مورد انتظـار دارای گسترش تهی هم باشد، در صورتی گسترش تهی انتخاب می شود که ترم پیش بینی متعلق به مجموعهٔ First ترم بعـدی آن در سمت راست قاعدهٔ مورد نظر باشد. مجموعه First ترم هایی که پس از ترم میانی A در سمت راست قواعد متفاوت ظـاهر می شوند را در اصطلاح مجموعهٔ پیرو یا Follow برای آن ترم میانی می‌گویند. برای نمونه در گرامر فوق:

Follow (Label) = First (Statement) ={S_Id, S_If, S_While}

اصولاً برای بدست آوردن مجموعه پیرو به روشهای زیر عمل می‌شود:

- در صورتی که ترم A در سمت راست یک قاعده به صورت زیر ظاهر شود

$B \rightarrow A\ \alpha$

آنگاه :

$Follow(A) = First(\alpha)$

- درصورتیکه ترم A در سمت راست یک قاعده به صورت زیر ظاهر شود

$B \rightarrow \alpha\ A$

آنگاه مجموعهٔ پیرو برای A شامل مجموعهٔ پیرو B می‌باشد

$Follow(B) \subseteq Follw(A)$

اما بالعکس صادق نیست. علت این است که هرجایی که B در سمت راست قواعد ظاهر شود می‌توان به جای آن αA را قرار داد اما، بالعکس صادق نیست.

- همواره در مجموعهٔ پیرو برای سرترم گرامر علامت خاتمهٔ فایل یا End of File وجود دارد. زیـرا ورودی تحلیلگـر نحوی یک فایل است. برای نمونه یک برنامه C را در نظر بگیرید. این برنامه به عنوان یک جملهٔ ورودی به کامپایلر داده می‌شود. چون برنامه در فایل است در انتهای آن علامت خاتمه فایل قرار می‌گیـرد. برنامـه ورودی در اینجـا از سرترم گرامر یعنی برنامه C مشتق می‌شود. پس بعد از سرترم گرامر علامت خاتمه فایل قرار می‌گیرد. بطور کلـی هر جمله ای که در ورودی کامپایلر قرار می‌گیرد باید از سرترم گرامر مربوطه مشتق شـود. چـون جملـه ورودی در داخل یک فایل است، لذا همواره در مجموعه پیرو سرترم گرامر، علامت خاتمه فایل وجود دارد. علامت خاتمه فایل را بطور اختصاصی معمولاً با $ مشخص می‌کنند.

بنابراین مشاهده می‌کنید چنانچه مجموعهٔ First برای یک ترم شامل عنصر تهی λ باشد، آنگاه بایـد عنصـر λ را از داخـل مجموعـه حذف و به جای آن عناصر مجموعهٔ پیرو را به داخل مجموعه First افزود. به این ترتیب:

$First(Label) = \{ id , \lambda \} = First(Label) + Follow(Label)$
$= \{ S\_Id , S\_If , S\_While \}$

مشاهده می‌کنید مجموعهٔ Follow برای یک ترم شاخص گسترش تهی برای آن ترم است. بنابراین می‌توان نتیجه گرفت:

یک گرامر LL(1) است اگر اشتراک مجموعه سرآغاز برای هر دو گسترش هر ترم متعلق به آن گرامر، تهی باشدو چنانچه آن ترم دارای گسترش تهی باشد اشتراک مجموعهٔ سرآغاز و پیرو آن ترم باید تهی باشد.

برای نمونه گرامر زیر LL(1) نیست:

LabeldSt ⟶ Label Statement
Label ⟶ id: | λ
Statement ⟶ AssignmentSt | ifSt | WhileSt
AssignmentSt ⟶ id = Expression
WhileSt ⟶ WHILE (Expression) Statement
ifSt ⟶ iF (Expression) Statement

علت LL(1) نبودن گرامر فوق این است که :

$First(Label) \cap Follow(Label) = \{S\_Id\}$

بنابر این با مشاهده Id در ورودی نمی توان تصمیم گرفت که آیا گسترش : $Label \rightarrow Id$ باید انتخاب شود یا گسترش $Label \rightarrow \lambda$.

برای بدست آوردن مجموعهٔ First می توان از ماتریس تجانس نیز استفاده نمود. برای نمونه به گرامر زیر توجه کنید :

P ⟶ DB| B
B ⟶ Ae
A ⟶ bS|S
S ⟶ aD| Ba
D ⟶ aD| d

چنانچه در حالت کلی ترم A با ترم B آغاز شود یک رابطه بطول یک بین ترم A یا B وجود دارد. می توان به ایـن ترتیـب یـک گـراف واسط ایجاد نمود. اکنون مسئله، به دست آوردن روابط بین هر دو گره در گراف است. یعنی هدف بدسـت آوردن وجـود روابـط بـا هـر

طولی بین هردو گره در داخل این گراف جهت دار روابط است. می توان از ماتریس تجانس استفاده نمود . به این ترتیب کـه اگـر بـین دوترم A وB یک رابطه آغاز شدن A توسط B وجود دارد، در سطر مربوط به A و ستون مربوط به B در ماتریس مقدار یـک قـرار داده می شود. به این ترتیب برای گرامر فوق ماتریس زیر ایجاد می شود :

	A	B	D	P	S	A	B	D	E
A	1	0	0	0	1	0	0	0	0
B	1	1	0	0	0	0	0	0	0
D	0	0	1	0	0	1	0	1	0
P	0	1	1	0	1	0	0	0	0
S	0	1	0	0	1	1	0	0	0
A	0	0	0	0	0	1	0	0	0
B	0	0	0	0	0	0	1	0	0
D	0	0	0	0	0	0	0	1	0
E	0	0	0	0	0	0	0	0	1

اکنون آنقدر ماتریس را در خودش باید ضرب نمود تا اینکه در دو تکرار متوالی مقادیر ماتریس با یکدیگر تفاوتی نکنـد. در ایـن صـورت ماتریس نهایی روابط First را مشخص می‌کند. باید توجه داشته باشید که در هنگام ضرب دو ماتریس Boolean بـه جـای ضـرب از عمل " و " منطقی یا AND استفاده می شود و به جای عمل جمع از OR منطقی استفاده می‌شود.

۴-۶- تبدیل گرامرها به فرم LL(1)

همانگونه که قبلاً نیز توضیح داده شـد اگـر هـر دو گسـترش متفـاوت هـر تـرم متعلـق بـه یـک گرامـر نهایتـاً بـا یـک تـرم مشـترک آغاز نشوند آن گرامر LL(1) اسـت. بـرای تبـدیل گرامـر بـه فـرم LL(1) بعضـاً مـی‌تـوان از روشـی موسـوم بـه فـاکتور گیـری چـپ استفاده نمود .

۴-۶-۱- فاکتورگیری از چپ

چنانچه در حالت کلی برای ترم Aگسترشهایی بصورت زیر موجود باشد:

$A \rightarrow a\,\alpha \mid a\,\beta \mid \gamma$
$First(a\,\alpha) \cap First(a\,\beta) = \{ a \}$

آنگاه:

با فاکتورگیری a از سمت چپ دو گسترش aβ و aα می‌توان گرامر را بصورت زیر بازسازی نمود:

$A \rightarrow a\,B \mid \gamma$
$B \rightarrow \alpha \mid \beta$

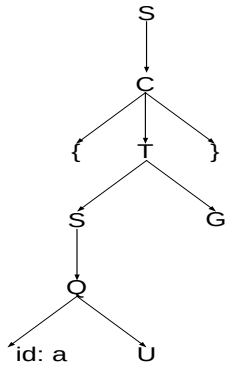
در فرم توسعه یافته می‌توان بدون استفاده از ترم کمکی B عمل فاکتور گیری را انجام داد و گسترشهای ترم میانی A را بصـورت زیـر تبدیل نمود :

$A \rightarrow a\,(\alpha \mid \beta) \mid \gamma$

با استفاده از روش فاکتور گیری چپ می‌توان قواعد مربوط به سر ترم s از گرامر ارایه شده بخش قبلی را بصورت زیر تبدیل نمود:

$S \rightarrow I \mid W \mid Q \mid C$
$Q \rightarrow id\ U$
$U \rightarrow :=No \mid \text{‘(‘M‘)’}$

به این ترتیب اکنون می‌توان بکار تولید درخت تجزیه که در بخش قبل با مشاهده id یعنی a در ورودی متوقف گردید ادامـه داد زیـرا،



حالا با دیدن ترم پایانی a در ورودی بلافاصله گسترش $Q \rightarrow id\ U$ انتخاب میشود و درخت تجزیه به این صورت تبدیل می‌گردد.

حالا با فراخوانی تحلیلگر لغوی در ورودی = : ظاهر می‌شود که در اینجا بطور قطعی می‌توان گفت که گسترش $U \rightarrow :=No$ باید انتخاب شود. در حالت کلی اگر در گرامر قواعد بصورت خود بازگشتی چپ وجود داشته باشد ، باز هم گرامر LL(1) نمی‌تواند باشد .

۴-۶-۲- تبدیل قواعد خود بازگشتی چپ

وجود قواعد بصورت خود بازگشتی چپ مغایر با LL(1) بودن گرامرها می‌باشد. برای نمونه به قواعد زیر توجه نمایید:

$A \rightarrow A\ \alpha \mid \beta$

جهت نقض LL(1) بودن گرامر کافی است اثبات شود که:

$First(A\ \alpha) \cap First(\beta) \neq \varnothing$

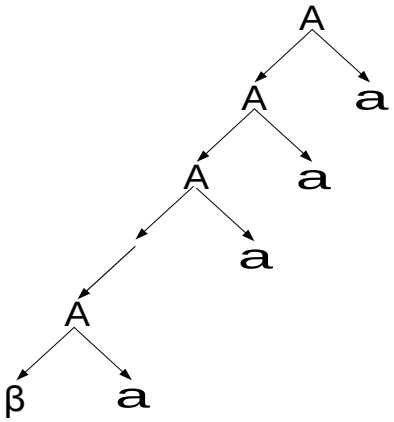
در اینجا واضح است که:

$first(A\ \alpha) \cap first(\beta) = \beta$

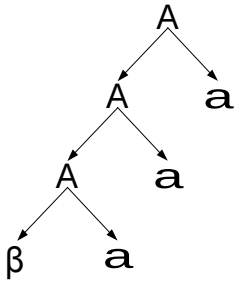
چرا که :

$first(A\ \alpha) = first(A) = first(\beta)$

بـــرای تبـــدیل قواعـــد مربـــوط بـــه A بصـــورت LL(1) بـــه ایـــن ترتیـــب میتـــوان اســـتدلال نمـــود کـــه دو قاعـــده $A \rightarrow A\alpha \mid \beta$ نمایانگر فرم کلی رشته هایی است که با β آغاز و با صفر یا بیشتر α ادامه می‌یابند. بنابر این هـر رشـته بـا فـرم کلـی $\beta\alpha\alpha\alpha\ldots\ldots\alpha$ از سر ترم A باید مشتق شود.



به این ترتیب برای رشته βααα درخت تجزیه بصورت زیر خواهد بود.

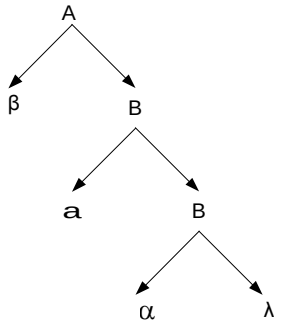


به همین ترتیب رشته β نیز از سر ترم گرامر مشتق می‌شود :

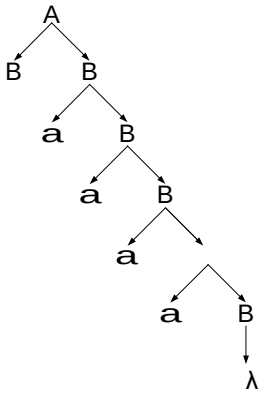
رشته های فوق را با گرامر زیر نیز می توان تولید نمود:

$A \rightarrow \beta\ B$
$B \rightarrow \alpha\ B \mid \lambda$

رشته βααα را در نظر بگیرید. این رشته بر طبق گرامر فوق از سر ترم A مشتق می شود.



و به همین ترتیب هر رشته با فرم کلی βααα....α از سر ترم A مشتق می‌شود. درخت تجزیه در حالت کلی بصورت زیر است.



بنابر این می‌توان نتیجه گرفت که برای حذف خود باز گشتی چپ از قواعد با فرم کلی :

$A \rightarrow A\ \alpha \mid \beta$

می‌توان آنها را با قواعد معادل زیر جایگزین نمود :

$A \rightarrow \beta\ B$
$B \rightarrow \alpha\ B \mid \lambda$

در فرم توسعه یافته گرامر بصورت زیر تبدیل می‌شود :

$A \rightarrow \beta\ \{\ \alpha\ \}$

در فرم توسعه یافته آکولاد به مفهوم تکرار صفر یا بیشتر است.

مثال گرامر عبارات را بصورت LL(1) تبدیل کنید.

E ⟶ E+T| E-T| T

T ⟶ T*F| T/F| F

F ⟶ id| No| (E)

به قواعد E توجه نمایید. دو گسترش E+T و E-T با یک ترم مشترک E آغاز می‌شوند. به همین ترتیب دو گسترش متفاوت ترم میانی T نیز با یک ترم آغاز می‌شوند. پس از انجام عمل فاکتور گیری چپ گرامر توسعه یافته عبارات بصورت زیر تبدیل می‌شود:

E ⟶ E (+|-)T| T

T ⟶ T (*|/)F| F

F ⟶ id| No| (E)

حالا با حذف خود باز گشتی چپ، گرامر بصورت زیر تبدیل می‌شود.

E⟶ T { (+|-) T}

T⟶ F {(*|/) F}

F⟶ id| No| (E)

در صورت وجود قواعد تهی با فرم کلی $A \rightarrow \lambda$ در گرامر نیز ممکن است گرامر LL(1) نباشد.

۴-۶-۳- حذف قواعد تهی

چنانچه در گرامری قواعد تهی وجود داشته باشد، آن گرامر ممکن است LL(1) نباشد. برای تبدیل گرامر به فرم LL(1) باید ابتدا قواعد تهی را حذف نمود و سپس با استفاده از روشهای فاکتور گیری و حذف خود بازگشتی چپ گرامر را به فرم LL(1) تبدیل نمود.

برای نمونه به گرامر زیر توجه نمایید:

LabelSt ⟶ Label Statement

Labet ⟶ id:| λ

Statement ⟶ Assignment | CallSt

AssignmentSt ⟶ id = Expression

CallSt ⟶ id '(' ActualParams ')' | id()

ActualParams ⟶ Expression| Expression.param

Expression ⟶ id| No

در گرامر فوق برای ترم میانی Label یک گسترش تهی وجود دارد. لذا باید

First (Label) ∩ Follow (label) = ∅
First (Label) = { Id }
Follow (label) = First(Statement) = First(AssignmentSt) + First(CallSt) ={ Id }
First (Label) ∩ Follow (Label) = { Id } ≠ ∅

بنابر این گرامر LL(1) نیست زیرا هنگامیکه انتظار مشاهده Label در ورودی می‌رود، با مشاهده id در ورودی نمی توان تصمیم گرفت که آیا گسترش' : Label → id 'باید انتخاب شود و یا گسترش Label →λ برای تبدیل گرامر فوق به فرم LL(1) باید گسترش Label → λ حذف شود و هر جایی که از ترم میانی Label در سمت چپ یک قاعده استفاده شده است، یک بار Label را تهی و بار دیگر بصورت غیر تهی در نظر گرفت. بنابر این گرامر بصورت زیر تبدیل می‌شود. :

LabelSt → Statement | Label Statement
Label → id

برای LabeledSt اکنون First(Statement) ∩ Follow(label) = {id } ≠ ∅ پس باید با استفاده از عمل فاکتورگیری چپ گرامر را به فرم LL(1) تبدیل نمود. برای این منظور باید ابتدا ترم ها را جایگزین کرد.

LabelSt → id = Expression

| id '(' ActualParams ')'

| id : Statement

پس از فاکتور گیری از id قاعده بصورت زیر تبدیل می‌شود:

LabeledSt → id St
St → = Expression
| '(' ActualParams ')'
| : Statement

در مورد Statement نیز باید عمل فاکتورگیری چپ را انجام داد:

Statement → Assignment | CallsSt

با جایگزینی Assignment و CallSt قاعده مربوط به Statement بصورت زیر تبدیل می‌شود:

Statement → id StTail
StTail → = Expression | id '(' ActualParams')'

تمرین : گرامر زیر را به فرم LL(1) تبدیل کنید.

S → L A B
L → d | λ
A → d A | B a
B → B b | λ

۴-۷- ایجاد جدول تجزیه بالا به پایین

جدول تجزیه ساختاری برای تولید برنامه تحلیلگر نحوی برای گرامرهای LL(1) است. برای نمونه به گرامر زیر توجه نمایید.

S→dAB I BaB
A→dA I Ba
B→bB | λ

برای این گرامر باید ابتدا مجموعه های سر آغاز را برای ترمهای میانی بدست آورد. با استفاده از این مجموعه ها جدول تجزیه بدست می‌آید:

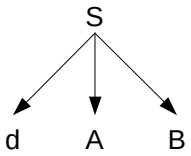
	a	b	d	$
S	BaB	BaB	dAB	
A	Ba	Ba	dA	
B		bB		

First (S) = {d} + First (B)
First (A) = {d} + First (B)
First (B) = {b} + Follow (B) ={b} +{a ,b,$ }

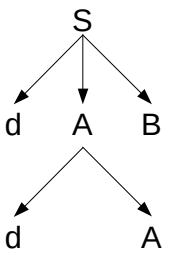
اکنون با استفاده از جدول تجزیه فوق براحتی می توان عمل تجزیه بالا به پایین را انجام داد. برای نمونه جمله ddabbb را با استفاده از جدول تجزیه فوق می توان براحتی تجزیه نمود.برای این منظور از یک پشته به نام پشته تجزیه استفاده می‌شود:

پشتـــة تجـــزيه	ورودي	قاعده
$S	_ddabbb$	S ⟶ dAB
$BAd	_ddabbb$	
$BA	_dabb$	A ⟶ dA
$BAd	_dabb$	
$BA	_abbb$	A ⟶ Ba
$BaB	_abbb$	B ⟶ λ
$Ba	_abbb$	
$B	_bbb$	B ⟶ bB
$Bb	_bbb$	
$B	_bb$	B ⟶ bB
$Bb	_bb$	
$B	_b$	B ⟶ bB
$Bb	_b$	
$B	$	B ⟶ λ

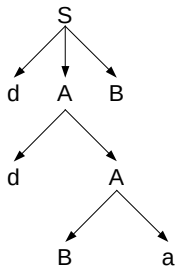
همانگونه که در جدول فوق مشاهده نمودید به انتهای رشته ورودی علامت خاتمه فایل یعنی $ افزوده می‌شود. عمل تجزیه از سر ترم گرامر آغاز می‌شود. و پس از سر ترم انتظار می‌رود که در ورودی علامت خاتمه فایل ورودی یعنی $ ظاهر شود. لذا بنابر این پیش بینی در آغاز کار رشته $S در داخل پشته قرار داده شده است.
عمل تجزیه از ردیف مربوط به سر ترم آغاز می‌شود با مشاهده اولین ترم در جمله $ddabbb ترم در ورودی که ترم پایانی d است . یه ستون d درسطر S در داخل جدول ارجاع می‌شود.
گسترش DaB در مکان (S,d) از جدول مشخص شده است لذا درخت تجزیه بصورت زیر ایجاد می‌شود



اکنون ترم بعدی از جمله $ddabbb خوانده می شود. این ترم پیش‌بینی نیز dاست . با توجه به اینکه ترم پیش‌بینی هنوز d است . طبق درخت تجزیه انتظار مشاهده A در ورودی می‌رود. در مکان (A,d)از جدول تجزیه گسترش dA قرار دارد .درخت تجزیه زیر حاصل می‌شود.

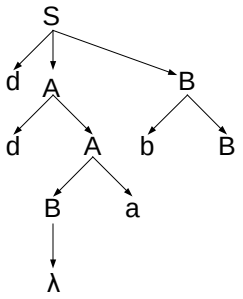


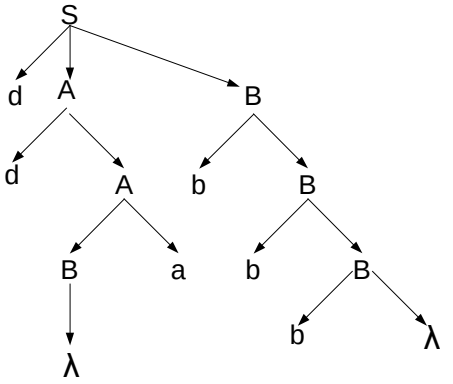
ورودی بعدی در جمله $dd<u>a</u>bbb ترم پایانی a است. طبق درخت تجزیه انتظار مشاهده A در ورودی می‌رود درمکان (A,a) از جدول گسترش Ba قرار گرفته است. بنابر این درخت تجزیه بصورت زیر توسعه داده می‌شود:



حالا با توجه به محتوی جدول تجزیه در مکان(B,a)باید گسترش $B\rightarrow\lambda$ انتخاب شود . بنابر این طبق درخت تجزیه انتظار دیدن a در ورودی می‌رود. در این لحظه ترم پیش بینی همان طوری که انتظار می‌رود
a است. ورودی بعدی در جمله dd<u>a</u>bbb ترم پایانی b ا ست . طبق درخت تجزیه انتظار مشاهده B در ورودی می‌رود. در مکان (Bو b) از جدول گسترش bB قرار گرفته است .بنابر این تا این مرحله درخت تجزیه بصورت زیر می باشد :

به همین ترتیب اگر عملیات ادامه پیدا کند نهایتا درخت تجزیه بصورت زیر ایجاد می شود:





مثال ـ گرامر عبارت زیر ارائه شده تبدیل به فرم LL(۱) نموده و جدول تجزیه برای یک تحلیلگر

پیش بینی کننده ایجاد کنید:

SimpleExp	⟶	**SimpleExp '+' Term | SimpleExp '-' Term | SimpleExp '||' Term | Term**
Term	⟶	**Term'/' Factor| Term'*' Factor|Term % Factor| Term && Factor|Factor**
Factor	⟶	**Number | ! Factor | '(' Expression ')'**
Expression	⟶	**Expression Relop SimpleExp | SimpleExp**
RelOp	⟶	**<|<=|!=|>|>=|==**

قواعد ارائه شده برای Expression دارای وضعیت خود بازگشتی چپ است :

Expression	⟶	**Expression Relop SimpleExp | SimpleExp**

در حالت کلی $A \rightarrow Aa \mid b$ را می توان با قواعد معادل زیر جایگزین کرد :

$A \rightarrow \beta B$
$B \rightarrow \alpha B \mid \lambda$

حالاExpression را مشابه A و SimpleExp Relop مشابه با aوSimpleExp را مشابه با β در نظر بگیرید.به این ترتیب قواعد مربوط به E بصورت زیر تبدیل می شوند :

Expression → SimpleExp E
E → Relop SimpleExp E | λ

در مورد SimpleExp ابتدا باید فاکتور گیری چپ نمود .

SimpleExp	⟶	**SimpleExp '+' Term | SimpleExp '-' Term | SimpleExp '||' Term | Term**

پس از فاکتور گیری چپ قواعد بصورت زیر تبدیل می‌شوند.

SimpleExp → SimpleExp Simple | Term
Simple → '+' Term | '-' Term | '||' Term

پس از حذف خود بازگشتی چپ قواعد بصورت زیر تبدیل می‌شوند :

SimpleExp → Term S
S → Simple S | λ
Simple → '+' Term | '-' Term | '||' Term

در قاعده مر بوط به S می توان Simple را با گسترش آن جایگزین نمود .بنا بر این قواعد فوق بصورت زیر ساده می‌شوند :

SimpleExp → Term S
S → (' + ' Term I ' – ' Term I '||' Term) S |λ

بطور خلا صه گرامر عبارات در فرم LL(1) بصورت زیر می باشد :

Expression	——	SimpleExp E
E	——	RelOp SimpleExp E | λ
RelOp	⟶	<| <=|!= | >| >=| ==
SimpleExp	⟶	Term S
S	⟶	(' + ' Term I ' – ' Term I '||' Term) S | λ
Term	⟶	Factor T
T	⟶	('/' Factor | '*' Factor | % Factor | && Factor) T | λ
Factor	⟶	Number| ! Factor| '(' Expression ')'

اکنون می توان مجموعه سرآغاز را برای ترمها محاسبه نمود :

First (Expression) = First(SimpleExp) = First(Term) = First(Factor) = {Number , ! , (}

First(E) = First(Relop))= {<,<=,!=,==,>=,>}

Follow (E) = Follow (Expression) = {')' ,$}

First(s) = {+,-,||}

Follow(s) = Follow(Simpleexp) = First(E)+Follow(E)={<.<=,!=,==,>=,>,')',$}

حالا با استفاده از روابط فوق می توان جدول تجزیه بالا به پایین را به سادگی ا یجاد کرد.

مثال ـ گرامر زیر را به فرم LL(1) تبدیل نموده و جدول تجزیه برای آن ایجاد کنید .

S → SA l BdA
A →Aa l b
B → ba l Bd l λ

تبدیل گرامر به فرم LL(1) در فرم توسعه یا فته بسیارساده تر است. لذا گرامر بصورت زیر تبدیل می‌شود.

S → BdA {A}
A → b {a}
B → [ba] {d}

اما چون λ#{d} = Follow(B) … First (B)گرامر LL(1) نمی‌با شد. بنا بر این بصورت زیر عمل باید نمود:

S → [ba] {d} dA {A}
A → b {a}

در مورد گسترش S, مشکل d{d} است. زیرا مشخص نیست که چه هنگامی می توان به d دومی رسید.

اما واضح است که d{d} = {d}d است. بنابر این گرامر بصورت زیر تبدیل می‌شود :

S → [ba] d {d} A {A}
A → b {a}

حالا می توان جهت تولید جدول تجزیه گرامر را به فرم ساده تبدیل نمـود. بـرای ایـن منظـور ابتـدا [ba] را بـا B , و d{d} را بـا C و A{A} را با D باید جایگزین نمود :

S→ BCD

A→bE

E→aE | λ

B→baB | λ

C→dF

F→dF | λ

D→AG

G→AG | λ

بنابراین برای ایجاد جدول تجزیه داریم

First(S) = First(B) = {b,d }
First(A) = {b}
First(B) = { b } +Follow(B) ={ b } +First(C) = {b,d}
First(C) = {d }
First(F) = {d,b}
First(D) = { b}
First(G)={b,$}

اکنون می‌توان جدول تجزیه را به سادگی برای این گرامر بدست آورد .

۸-۴- تجزیه گرهای کاهینه بازگشتی

تجزیه گرهای کاهینه بازگشتی یا Recursive descent parser را می توان برای گرامرهای LL(1) مورد استفاده قرار داد. در این روش برای هر ترم میانی و سر ترم یک روال یا زیر برنامه به همان نام ایجاد می‌شود. به این ترتیب قواعد گرامر را عیناً با استفاده از توابع خود بازگشتی تبدیل به کد بر نامه می نمایید. لذا مزیت این روش سادگی ایجاد و خوانایی کد برنامه تجزیه‌گر است.
گرامر زیر بخشی از گرامر زبان c است که ساختار دستورات مربوط به تعریف متغیرها را نشان می‌دهد:

VarDef →Def;VarDef | Def;
Def →Type VarList
Type → int | float | char | double
VarList → id,VarList | id

این گرامر LL(1) نیست(چرا؟) و باید به LL(1) تبدیل گردد. نتیجه گرامر زیر خواهد بود:

VarDef →Def;A
A → VarDef | λ
Def →Type VarList
Type → int | float | char | double
VarList → id,B
B → VarList | λ

در ساختار کلی یا در واقع بر نامه اصلی برای این نوع تحلیلگر پس از انجام عملیات اولیه به نام init تحلیلگر لغوی به نام NextSymbol فراخوانده می شود تا نوع اولین ترم پیش بینی در یک متغیر سراسری به نام CurrentSymbol قرار گیرد. سپس روال تعیین شده برای سر ترم گرامر یعنی VarDef مورد فراخوانی قرار می‌گیرد. بنا بر این بدنه اصلی برنامه تحلیلگر کاهینه بازگشتی بصورت زیر است:

```
void main(){
    Init();
    NextSymbol();
    VarDef ();
}
```

Current Symbol متغیری سراسری از نوع شمارش پذیر Symbols است که قبل از این برای تعیین نوع لغات توسط تحلیلگر لغوی مشخص شد. نوع Symbol حاوی انواع لغات موجود در زبان مورد نظر است. برای گرامر فوق بصورت زیر تعریف می‌شود:

```
enum   Symbols{S_Semi, S_Int, S_Float, S_Char, S_Double, S_Id,S_Comma };
Symbols CurrentSymbol;
```

پس از اینکه تحلیلگر لغوی اولین لغت را از بر نامه ورودی تشخیص و در داخل Curent Symbol قرار داد می بایست روال VarDef فراخوانی شود. در واقع VarDef نام سر ترم گرامر است چرا که در این روش برای هر ترم میانی و سر ترم روالی به همان نام نوشته می شود. روال VarDef بر اساس قاعده مربوطه نوشته شده است:

```
void VarDef(){                    // روال تشخیص سر ترم گرامر
    Def() ;                       // فراخوانی روال ترم میانی Def
    Expect  (S _ Semicolon ) ;    // انتظار مشاهده S_Semicolon در ورودی می رود
    A() ;                         // فراخوانی روال ترم میانی A
  }
```

همانطوری که مشاهده می‌شود اکنون در ابتدای روال VarDef باید روال Def فراخوانی گردد. سپس بنا بر قاعده مربوطه انتظار مشاهده لغت از نوع Semicolon می‌رود برای این منظور تابع Expect با پارامتر S_Semicolon موردفراخوانی قرار گرفته است. بنابر این محتوی CurrentSymbol در داخل Expect بنابر گرامر با S _ Semicolon مقایسه می‌شود. کد این روال بصورت زیر است:

```
void Expect ( S: Symbols){
      If  (CurrentSymbol == S)//  اگر ترم پیش بینی مساوی با پارامتر ارسالی است
            NextSymbol();          //  آنگاه لغت بعدی بعنوان ترم پیش بینی خوانده شود
      else   Syntaxerror();        //  وگرنه پیام خطای نحوی صادر شود
     }
```

همچنین زیر برنامه‌های Syntaxerror, NextSymbol بصورت زیر پیاده‌سازی می‌گردند:

```
void NextSymbol(){
      CurrentSymbol=lex();      // زیربرنامه تحلیلگر لغوی فراخوانی شده و نوع لغت بعدی را
     }                          //بعنوان نتیجه به ما می‌دهد
void Syntaxerror(){
  cout<<"syntax error ";
  exit(0);
}
```

بنابراین قانون کلی برای مشاهده هر ترم میانی باید روال مربوط به آن فراخوانی شود و برای هر ترم پایانی باید با زیربرنامه Expect آن را مشاهده نماییم. با توجه به جدول تجزیه بالا به پایین گرامر فوق سایر روالهای تحلیلگر کاهینه بازگشتی برای گرامر فوق بصورت زیر خواهد بود:

	;	,	$	id	int	float	char	double
VarDef	-	-	-	-	Def;A	Def;A	Def;A	Def;A
A	-	-	λ	-	VarDef	VarDef	VarDef	VarDef
Def	-	-	-	-	Type VarList	Type VarList	Type VarList	Type VarList
Type	-	-	-	-	int	float	char	double
VarList	-	-	-	id,B	-	-	-	-
B	λ	-	-	VarList	-	-	-	-

جدول تجزیه گرامرفوق

```
void A(){
 switch(CurrentSymbol){
  case S_Int:
  case S_Float:
  case S_Char:
  case S_Double:VarDef();
                  break;
  case S_Eof: return;
  default: Syntaxerror();
}
}
```

```
void Def(){
        Type();
        VarList();
}
```

```
void Type(){
switch(CurrentSymbol){
  case S_Int:
  case S_Float:
  case S_Char:
  case S_Double:NextSymbol();
                  break;
  default: Syntaxerror();
}
}
```

```
void VarList(){
  Expect(S_Id);
  Expect(S_Comma);
  B();
}
```

```
void B(){
   if(CurrentSymbol==S_Id)
        VarList()
}
```

A → VarDef | λ
Def →Type VarList
Type → int | float | char | double
VarList → id,B
B → VarList | λ

در بخش قبلی گرامر عبارات به فرم LL(1) تبدیل شد تا بتوان جدول تجزیه برای تحلیلگر نحوی پیش‌بینی کننده برای تشخیص عبارات ایجاد نمود. برای ایجاد تحلیلگر کاهینه بازگشتی بهتر است که گرامر در فرم توسعه یافته به صورت LL(1) تبدیل شود. گرامر عبارات در فرم توسعه یافته به صورت زیر است:

```
Expression → SimpleExp{ RelOp SimpleExp }
RelOp      → <| <=| <>| >| >=| IN
SimpleExp  → Term{ ( '+'| '-'| Or) Term}
Term       → Factor{ ( '/'| '*'| DIV| AND ) Fator}
Factor     → Number| NOT Factor| '(' Expression ')'| Variable
```

Variable → identifier { '[' Dim ']' }

Dim → Expression { ',' Expression }

حالا می‌توان براساس گرامر فوق تجزیه‌گر کاهینه بازگشتی را به صورت زیر ایجاد کرد

```
Begin init; NextSymbol; Expression; End;
(* Expression → SimpleExp{ RelOp SimpleExp} *)
  Procedure Expression;
        Begin
           SimpleExp;
          While CurrentSymbol in First(RelOp) do
                  Begin
                    RelOp;
                    SimpleExp;
                  End;
            End;
(* RelOp → < | <= | = | <> | >= | > | IN *)
Procedure RelOp;
Begin
  If CurrentSymbol in [S_Lt, S_Le, S_Eq, S_Ne, S_Ge,
    S_Gt] then NextSymbol else Expect(S_In);
End;

(* SimpleExp → Term {('+'| '-'| Or)Term}*)
Procedure SimpleExp;
Begin
  Term;
  While CurrentSymbol in [S_Add, S_Sub, S_Or] do
  Begin
    NextSymbol;
    Term;
  End;
End;

(* Term → Factor { ('/'| '*'| DIV| AND) Factor} *)
Procedure Term;
Begin
  Factor;
  While CurrentSymbol in [S_Div, S_Mul, S_IntDiv,
  S_And] do
  Begin
    NextSymbol;
    Factor;
  End;
End;

(* Variable → Identifier {'[' Dim ']'} *)
Procedure Term;
Begin
  Expect(S_Id);
  While CurrentSymbol = S_OpenSquareBracket do
  Begin
    NextSymbol;
    Dim;
    Expect(S_CloseSquareBracket);
  End;
End;
```

۴-۹- بهبود از خطا

یکی از دامنه‌های تحقیقاتی بسیار وسیع در زمینه کامپایلر که حتی امروزه به هوش مصنوعی هم ارتباط پیدا کرده است، مسأله بهبود از خطاست. مسأله اینجاست که یک کامپایلر قادر باشد پس از مشاهده خطای نحوی در متن برنامه داده شده به درستی به کار خود ادامه دهد و حداکثر تعداد خطا را در یک مرحله از کامپایل تشخیص دهد. نکته این است که برای نمونه اگر در برنامه‌ای به جای if برنامه نویس اشتباهاً uf وارد کند، کامپایلر با مشاهده لغت uf که یک شناسه است به جای if، گمراه نشده و پیام‌های خطای اضافی صادر نکند و قادر باشد که در یک مرحله از کامپایل حداقل تعداد پیام لازم برای حداکثر تعداد خطا را ایجاد نماید.

برای درک بهتر مسأله بهبود از خطا، در زیر یک مثال ارایه می‌شود. برای این منظور روال‌های زیر را که قبلاً جهت تشخیص ثابت‌ها ارایه شده بود، در نظر بگیرید:

```
(* ConstantDefPart → Const ConstantDef
  {ConstantDef} *)
Procedure ConstantDefPart;
Begin
  NextSymbol;
  ConstantDef;
  While CurrentSymbol = S_Id do ConstantDef;
End;

(* ConstantDef → id = (NO| id); *)
Procedure ConstantDef;
Begin
  Expect(S_Id);
  Expect(S_Eq);
  If CurrentSymbol = S_NO then NextSymbol else
    Expect(S_Id);
  Expect(S_Semicolon);
End;
```

اکنون به قطعه کد زیر که دارای خطای نحوی است توجه نمایید:

```
Const
     a := 5;
     b =  ;
     c = 8;
```

برای تجزیه کد فوق روال ConstantDef فراخوانی می‌شود. در ضمن عمل تجزیه هنگامی که پس از ترم پایانی a در ورودی انتظار مشاهده S_Eq می‌رود، برخلاف انتظار لغت =: از نوع S_Assign ظاهر می‌شود. بنابراین در داخل Expect روال SyntaxError فراخوانی می‌شود. در اینجا نکته چگونگی عملکرد SyntaxError است. اگر SyntaxError به صورت زیر نوشته می‌شود:

```
Procedure SyntaxError;
Begin
  Error('Syntax', LineNo, ColNo);
End;
```

چون در این حالت NextSymbol در داخل Expect مورد فراخوانی قرار نمی‌گیرد، محتوای CurrentSymbol همان S_Assign باقی خواهد ماند. پس از اینکه پیغام خطا در سطر LineNo و ستون ColNo از متن برنامه مورد کامپایل اعلام شد، کنترل به روال Expect و پس از آن به فراخواننده Expect یعنی ConstantDef می‌رسد. اما تغییر نکردن مقدار ترم پیش‌بینی یعنی مقدار CurrentSymbol موجب می‌شود که به غلط پیام خطای دومی اعلام می‌شود. به این ترتیب زمانی که دستورالعمل Expect(S_No) اجرا می‌شود، چون محتوای CurrentSymbol مساوی با S_No نبوده و مساوی با S_Assign است لذا مجدداً به غلط پیام خطای دیگری صادر می‌شود و به همین ترتیب پیام‌های خطای بعدی یکی پس از دیگری صادر می‌شوند. اگر NextSymbol به صورت زیر در داخل SyntaxError فراخوانی شود:

```
Procedure SyntaxError;
```

```
Begin
  Error('Syntax', LineNo, ColNo);
  NextSymbol;
End;
```

آنگاه پس از اعلام پیام خطا، لغت بعدی یعنی عدد 5 که از نوع S_No است، از ورودی خوانده می‌شد. به این ترتیب با تغییر CurrentSymbol به S_No، روال ConstantDef به درستی می‌توانست به کار خود ادامه دهد. با وجود این، افزایش NextSymbol یا به عبارت دیگر چشم پوشی از مورد خطا که در اینجا S_Assign بود همواره مشکل گشا نیست. برای مثال در سطر بعدی یعنی ; =b پس از تشخیص = یا S_Eq محتوای CurrentSymbol به غلط S_Semicolon خواهد بود. و در زمانی که دستورالعمل Expect(S_No) اجرا می‌شود، محتوای CurrenSymbol برابر S_Semicolon است و در اینجا چشم‌پوشی از این لغت یعنی S_Semilcolon موجب خطا می‌شود و بهتر است که NextSymbol از روال SyntaxError حذف شود.
چه بایدکرد؟ مشاهده نمودید که در یک مورد بودن NextSymbol در داخل SyntaxError برای چشم‌پوشی ازخطای نحوی و در واقع لغت غیرقابل انتظار، ضروری است و در موارد دیگر این عمل غلط می‌باشد. برای رفع این مشکل در داخل هر قاعده برای هر ترم به طور مجزا مجموعه‌ای از ترم‌ها که پس از آن ترم در قاعده‌ای ظاهر می‌شوند را به عنوان مجموعهٔ Stop یا متوقف کننده در نظر گرفته می‌شود. اکنون در داخل SyntaxError آنقدر ترم‌های بعدی خوانده شده و از آن‌ها چشم‌پوشی می‌شود تا طبق قاعده بتوان به یکی از ترم‌های بعدی مورد انتظار در داخل مجموعهٔ Stop رسید. برای نمونه طبق گرامر در داخل ConstantDef پس از S_Id انتظار دیدن S_Eq و پس از آن انتظار S_Semicolon در ورودی می‌رود، پس از S_Semicolon مسلماً باید ترم‌هایی که پس از ConstantDef می‌توانند ظاهر شوند، قرار گیرند. این ترم‌ها شامل S_Id که شروع کننده ConstantDef دیگری است و همین طور ترم‌هایی که بعد از خود ConstantDefPart می‌توانند ظاهر شوند، است. به این ترتیب روال‌ها را می‌توان به صورت زیر بازنویسی کرد:

```
(* ConstantDefPart → Const ConstantDef,
  {ConstantDef} *)
Procedure ConstantDefPart(Stop: set of Symbols);
Begin
  NextSymbol;
  ConstantDef([S_Id]+stop);
  While CurrentSymbol = S_Id do
  ConstantDef([S_Id]+stop);
End;

(* ConstantDef → Id '=' (NO| Id ':') *)
Procedure ConstantDef(Stop: set of symbols);
Begin
  Expect(S_Id, [S_Eq, S_No, S_Id,
    S_Semicolon]+stop);
  Expect(S_Eq,[S_No, S_Id, S_Semicolon]+stop);
  If CurrentSymbol = S_NO then NextSymbol
    Else expect(S_id, stop+[s_Semicolon]);
  Expect(S_Semicolon, Stop);
End;

Procedure Expect(S:Symbols,Stop:set of symbols);
Begin
  If CurrentSymbol = S then NextSymbol
    Else SyntaxError(Stop);
End;

Procedure SyntaxError(stop: set of symbols);
```

```
Begin
  Error(Syntax, LineNo, ColNO);
  While not(CurrentSymbol in stop) do NextSymbol;
End;
```

در حالت کلی واضح است که برای محاسبهٔ مجموعه stop به صورت زیر عمل می‌شود:

الف- چنانچه E → e1 e2 e3 …… en آنگاه:

$$\begin{matrix} i=1...n-1 & j=(i+1)...n \\ Stop(e_i) = \Sigma First(e_j) + stop(E) \\ stop(e_n) = stop(E) \end{matrix}$$

ب- چنانچه E → e1 | e2 | …… | en آنگاه:

$$Stop(e_i) = stop(E)$$
$$i=1...n$$

ج- چنانچه E → {e1} آنگاه:

$$Stop(e_1) = stop(E) + First(e_1)$$

به این ترتیب برنامه تحلیلگر لغوی که قبل از این نوشته شده بود به صورت زیر تبدیل می‌شود:

```
Begin
  Init;              //عملیات مقدماتی
  NextSymbol;        //گرفتن لغت بعدی از تحلیلگر لغوی
  ProgramX([S_Eof]); //انتظار سرترم و علامت خاتمه فایل پس از آن
End.
```

۴-۱۰- مولد تحلیلگر نحوی

مولد تحلیلگر نحوی برنامه‌ای است که گرامر یک زبان را دریافت نموده و برای آن یک تحلیلگر نحوی یا تجزیه‌گر ایجاد می‌نماید در این قسمت نوعی خاص از مولد ارایه شده است که گرامر زبان را در داخل یک فایل می پذیرد و بر اساس گرامر عمل، تحلیل نحوی را انجام می‌دهد. این مولد در واقع یک برنامه پیمایش گراف است. هر قاعده در گرامر LL(1) را می‌توان به عنوان یک گراف در نظر گرفت.

دربرنامه مولد تحلیلگر نحوی که در زیر ارائه شده، گرامر درون یک فایل متن یا در اصطلاح Text بنام grammer.txt قراردارد. برای قرار دادن گرامر در داخل این فایل در هر سطر ابتدا ترم سمت چپ قاعده و بلافاصله گسترش آن مشخص می‌شود. برای سادگی و خواناتر شدن برنامه فرض شده که هرترم میانی و سرترم با یک حرف لاتین درشت و هر ترم پایانی با یک حرف کوچک مشخص شده است. به این ترتیب برای نمونه گرامر ساده:

S → cA | BdS
A → aAB | d
B → bB | d

به صورت زیر در فایل Grammer.txt ذخیره می شود:

```
6
ScA
SBdS
AaAB
```

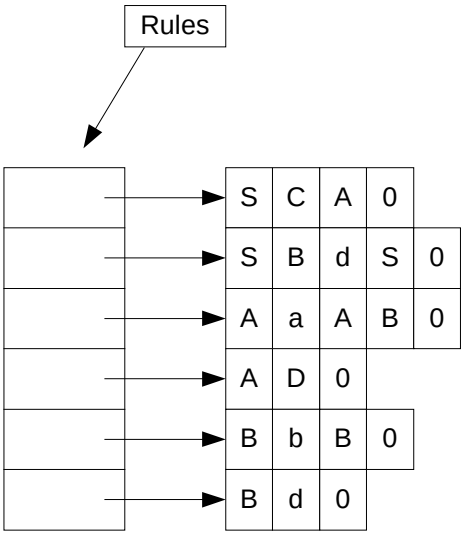
Ad
BbB
Bd

در اولین سطر از فایل grammer.txt تعداد قواعد یا در واقع تعداد سطرهای فایل مشخص شده است. بدنه برنامه اصلی main در زیر ارایه شده است. باز هم به خاطر سادگی و خوانایی برنامه، متن برنامه مورد کامپایل در داخل یک رشته بـه نـام Statement در نظـر گرفته شده است.

```
Void main()
int noRules,  //تعداد قواعد در گرامراصلی
len; //تعدادترمها در جمله ورودی
char **rules, //ماتریس قواعد گرامر

*Statemen  //جمله ورودی کامپایل
rules1  تولید ماتریس قواعد گرامر در آرایه دو بعدی
NoRules=ReadGrammer(&rules,statement);
Rules  //تحلیل نحوی جمله داده شده statement بااستفاده از ماتریس قواعد
Len = TopDownParse(rules,statement,0,NoRules,0);
If (len!=strlen(statement}((clrscr;puts3//;")
//اعلام پیام خطا در صورتیکه کار تحلیل نحوی تا آخر جمله داده شده ادامه پیدانکرده باشد (خطای نحوی)
```

دراولین سطر از برنامه اصلی تابع ReadGrammer فراخوانی می‌شود. ایـن تـابع گرامـر را از داخـل فایـل grammer.txt خوانـده، درداخل یک آرایه از نشانگرها به آرایه‌ها قرار می‌دهد. آدرس شروع آرایه نشانگرها در مکان مشخص شده توسط پـارامتر Pord از تـابع ظاهر می‌شود. هر آرایه حاوی یک قاعده گرامر خواهد بود. به عنوان نمونه برای گرامری که در ابتدای این بخش ارائه شد، ساختار آرایه



کهRules نام دارد به صورت زیر است:

کد تابعReadGrammer در زیر ارایه شده است.

int ReadGrammer(char *pord,char *statement)

گرامر را از فایل ورودی به ماتریس مربوطه انتقال می‌دهد. از انتهای فایل جمله ورودی را می‌خواند //

```
File *fp;
Char line[100],**rules;
Int I,NoRules;
```

```
Fp=fopen('Grammer2.txt','rt');
If (!fp) fp=stdin;
If (fp==stdin)
{clrscr(); printf('No Rules');
fscanf(fp%',d,&NoRules);
ایجاد آرایه از نشانگرها بطول تعداد قواعدگرامر به اضافه یک//
rules=(char**)malloc((NoRules+1)*sizeof(char));
rules[NoRules]=NULL;
for(i=0;!feof(fp)&& i<NoRules;i++)
قرار دادن قواعددرون ماتریس //}
fscanf(fp%',line);
rules[i]=malloc(strlen(line)+1);
strcpy(rules[i],line);
{
خواندن جمله مورد کمپایل//
pord=rules; // برگرداندن آدرس ماتریس قواعد
return NoRules; // برگرداندن تعداد قواعد به عنوان خروجی تابع
}
```

اکنون با تشکیل ماتریس قواعد در Rules و خواندن جمله مورد کامپایل در رشته statemwnt می‌توان با فراخوانی تابع TopDownParseعمل تحلیل نحوی را انجام داد. این تابع برای انجام عمل تحلیل نحوی از Rules[start] شروع می‌کند. در آغاز کار مقدار Start مساوی با صفر است، لذا در آغاز کار تحلیلگر نحوی با سرترم گرامر آغاز می‌شود. ترم پیش‌بینی برای این تابع در Statement[ix] قرار گرفته است.

تابع تحلیلگر نحوی//

```
int TopDownParse(char** rules, char* statement, int
  start, int NoRules, int ix)
پارامتر Rules حاوی قواعد گرامر است.//
//پارامتر start حاوی قاعده ترم میانی مورد انتظار است.
//پارامتر ix حاوی اندیس ترم پیش‌بینی در جمله مورد انتظار است.
int i, j, k, m, NewK;
char leftterm, input;
//1-Match the leftmost
input=statement[ix]; //input تعیین ترم پیش‌بینی در متغیر
// با فراخوانی تابع StartWhichRules مقدار I نشان می‌دهد که در داخل
// آرایه Rules کدام گسترش از گسترش‌های ترم مورد انتظار
//یعنی rules[start] با ترم پیش‌بینی input آغاز می‌شود.//
I = StartWhichRule(rules, input, start, NoRules);
if(i<0) return ix;
//rules[i][0] پیمایش سمت راست قاعده
for(j=1;k=ix;rules[i][j]&&statement[k]&&i<NoRules;j++);
// با فراخوانی تابع StartWhichRules مقدار I نشان می‌دهد که در داخل
// آرایه Rules کدام گسترش از گسترش‌های ترم مورد انتظار
//یعنی rules[start] با ترم پیش‌بینی input آغاز می‌شود.//
if(isupper(rules[i][j])
{
//جستجو برای یافتن قاعده مربوط به ترم میانی ظاهرشده در ضمن پیمایش سمت راست
for(m=0;rules[m][0]!=rules[i][j]&& m<NoRules;m++);
```

```
NewK=TopDownParse(rules,statement,m,NoRules,k);
If(k= =NewK) return 0;else k=NewK};
                                        وگرنه ترم بعدی در سمت راست قاعده یک ترم پایانی است//
else //خواندن ترم پایانی بعدی در ورودی
if(rules[i][j]= =statement[k]) k++;
return k;
}
```

با فراخوانی تابع StartWhichRules مقدار I نشان می‌دهد که در داخل آرایه Rules کدام گسترش از گسترش‌های ترم مورد انتظار یعنی [rules[start با ترم پیش‌بینی input آغاز می‌شود. با فراخوانی تابع StartWhichRules مشخص می‌شود که ترم پیش‌بینی input کدام گسترش ازگسترش‌های ترم مورد نظر یعنی [rules[start][0 را آغاز می‌کند. در صورت یافتن گسترش مورد نظر شماره اندیس قاعده آن در متغیر i برگردانده می‌شود. به این ترتیب [rules[i][0 باید مساوی با [rules [start][0 باشد. با یافتن یک قاعده مناسب، سمت راست آن قاعده مورد پیمایش قرار می‌گیرد. در ضمن پیمایش چنانچه تحلیلگر به یک ترم میانی برسد، با فراخوانی خود به طور خودبازگشتی بر مبنای آن ترم میانی عمل تحلیل نحوی را ادامه می‌دهد، پس از این فراخوانی NewK نمایانگر اندیس ترم پیش‌بینی در جمله ورودی یعنی statement است. واضح است که مقدار این اندیس باید متفاوت از اندیس قبل از فراخوانی خودبازگشتی تابع یعنی K باشد.

تابع StartWhichRule مشخص می‌کند، که ترم پیشبینی input آغاز کننده کدام گسترش از گسترش‌های متفاوت ترم [rules[start][0 است. برای این منظور چنانچه LeftNonTerm مساوی با [rules[start][0 قرار داده شود، تحلیلگر به دنبال گسترشی از این ترم میانی است یا به عبارت دیگر به دنبال یک [rules[I می‌گردد. که اولاً [rules[i][0 مساوی با LeftNonTerm باشد و ثانیاً ترم [rulse[i][1 یا ترم مورد پیش‌بینی یعنی input آغاز می‌شود. اگر[rulse[i][1 یک ترم میانی باشد، تابع به صورت خود بازگشتی فراخوانی می‌شود تا مشخص شود آیا آن ترم میانی با ترم پیش‌بینی یعنی input آغاز می شود. اگر موفق نشد به سراغ گسترش دیگر ترم میانی می‌رود در این مرحله می‌توان با استفاده از مجموعه سرآغاز کار تحلیلگر را تسریع نمود. برنامه کامل مولد تحلیلگر نحوی در زیر ارایه شده است.

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>
#include<malloc.h>
#include<ctype.h>
int ReadGrammer(char **pord,char*statement)
{
FILE *fp;
Char line[100],**rules;
Int i,NoRules;
Fp=fopen(“Grammer2.txt”,”rt”);
If(!fp) fp=stdin;
If(fp==stdin)
{clrscr(); printf(“No Rules”);}
fscanf(fp%,”d”,&NoRules);
rules=(char **) malloc((NoRules+1)*sizeof(char *));
rules[NoRules]=NULL;
for(i=0;!feof(fp)&&i<NoRules;I++);
{
fscanf(fp%,”s”,line);
rules[i]=malloc(strlen(line)+1);
strcpy(rules[i],line);
}
if(!feof(fp))fscanf(fp%,”s”,statement);
*pord=rules;
return NoRules;
```

```
}//End of ReadGrammer
//This function matches the leftmost term with input
int StartWhichRule(char **rules,char input,int start,int NoRules)
{int success,I,k;
char LeftNonTerm;
LeftNonTerm=rules[start][0];
//Look for a left terminal in the production rule
i=start;
while((rules[i][1]!=input)&&(rules= =LeftNonTerm)) I++;
//Hint:here is a catch ----ERORR----Find it out
if(rules[i][1]==input)
return I;
//Look for a left nonterminal in the production rule
for(i=start;success=-1;rules[i][0]= =LeftNonTerm &&success<0;I++)
if(isupper(rules[i][1]))
{for(k=0;rules[k][0]!=rules[i][1]&&k<NoRules;k++);
success=StartWhichRules(rules,input,k,NoRules);
}
if(success<0) return success;
return I-1;
}
int TopDownParse(char **rules,char *statement,int start,int NoRules,int ix)
{
int i,j,k,m,NewK;
char leftterm,input;
//Match the leftmost
input=statement[ix];
i=StartWhichRule(rules,input,start,NoRules);
if (i<0) return ix;
for(j=1,k=ix,rules[i][j]&&statement[k]&&i<NoRules;j++)
if(isupper(rules[i][j]))
{for(m=0;rules[m][0]!=rules[i][j]&&m<NoRules;m++);
newK=TopDownParse(rules,statement,m,NoRules,k);
if(k==NewK) return 0; else k=NewK};
else if(rules[i][j]==statement[k])k++;
return k;
}

void main()
{int NoRules,len;
char **rules,*statement;
NoRules=ReadGrammer(&rules,statement);
Len=TopDownParse(rules,statement,0,NoRules,0);
If(len!=strlen(statement)){clrscr(); puts("ERROR");}
}
```

۴-۱۱- پرسش و پاسخ

۴-۱ : تولید الگوریتم تحلیل نحوی بر مبنای عملکرد ذهن به چه صورت انجام میگیرد ؟

۴-۲ : LookAhead چیست ؟

۴-۳ : مجموعه های سرآغازو پیرو ی یک ترم را تعریف کنید و به چه گرامری LL(1) می‌گوییم ؟

۴-۴ : حذف قواعد تهی به چه صورت انجام میگیرد ؟

۴-۵ : تجزیه گرهای کاهینه بازگشتی به چه صورت عمل میکنند ؟

۶-۴ : منظور از بهبود از خطا در یک کامپایلر چیست ؟

۴-۱۲- تمرین

تمرین ۱– گرامر زیر را برای ساختار لیست در نظر بگیرید:

$S \rightarrow (L) \mid a$
$L \rightarrow LS \mid S$

این گرامر را به فرم LL(1) تبدیل نموده برای آن جدول تجزیه بالا به پایین ایجاد کنید. با استفاده از این جدول تجزیه، جمله (a,(a,a)) را مورد تحلیل نحوی قرار دهید.

تمرین ۲- گرامر زیر را به فرم LL(1) تبدیل نموده جدول تجزیه برای یک تحلیلگر پیش‌بینی کننده ایجاد نمایید.

$S \rightarrow SAB \mid AB$
$A \rightarrow Aa \mid a$
$B \rightarrow Bb \mid l$

تمرین ۳- یک الگوریتم برای تحلیلگر پیش‌بینی کننده ایجاد کنید که با استفاده از جدول تجزیه و یک پشته تجزیه عمل تحلیل نحوی را انجام دهد. برای این الگوریتم یک تابع به صورت زیر ایجاد کنید:

```
int PredictiveParser(char** ParseTable, int NoRows, int Nocols)
```

تمرین ۴- یک مولد تحلیلگر پیش‌بینی کننده ایجاد کنید که گرامر LL(1) را در ورودی می‌پذیرد. این مولد سپس، مجموعه‌های سرآغاز برای ترم‌های میانی را محاسبه می‌کند. با استفاده از مجموعه‌های سرآغاز به راحتی می‌توان جدول تجزیه را تولید کرد.

تمرین ۵- گرامر زیر را به فرم توسعه یافته LL(1) تبدیل نموده، یک تحلیلگر کاهینه بازگشتی برای آن ایجادکنید. در ضمن مسأله بهبود از خطا را نیز در نظر بگیرید.

$S \rightarrow SbB \mid SaB \mid LaA$
$L \rightarrow LaB \mid LbB \mid l$
$A \rightarrow bA \mid d$
$B \rightarrow Bb \mid l$

تمرین ۶ – گرامر ارایه شده در تمرین ۵ را تبدیل به فرم LL(1) نموده و سپس جدول تجزیه بالا به پایین برای آن ایجاد کنید.

تمرین ۷ – گرامر زیر را به فرم LL(1) تبدیل کنید:

$A \rightarrow a$
$A \rightarrow A1\ a1$
$A \rightarrow j\ A2\ a2$
… …
… ….
$An \rightarrow A\ an+1$

تمرین ۸ – یک الگوریتم کلی برای حذف گسترش‌های تهی در گرامر زبان‌ها ارایه دهید.

تمرین ۹ – گرامر زیر را به فرم LL(1) تبدیل نموده یک تجزیه‌گر کاهینه بازگشتی برای آن ایجاد کنید.
مسأله بهبود از خطا را نیز در نظر بگیرید.

S → SAB | Bda
A → BdA | dBa
B → Bb | l

تمرین ۱۰– چگونه می‌توان برای یک تجزیه‌گر پیش‌بینی کننده مسأله بهبود از خطا را در نظر گرفت. الگوریتم خواسته شده در تمرین ۳ را با در نظر گرفتن مسأله بهبود از خطا تکمیل کنید.

تمرین ۱۱ – برنامه یک تجزیه‌گر کاهینه بازگشتی برای گرامر یکی از زبان‌های رایج برنامه نویسی ایجاد کنید. مسأله بهبود از خطـا در نظر گرفته شود.

تمرین ۱۲- مولد تحلیلگر نحوی ارایه شده در انتهای فصل را آنچنان تکمیل نمایید که تحلیلگر لغوی را فراخوانی کند تا ترم بعـدی را از ورودی دریافت کند. ترم‌های میانی نیز به هر صورتی در گرامر ظاهر شوند. با محاسبه اتوماتیک مجموعه First برای ترم‌های میانی و سرترم می‌توان کار مولد تحلیلگر را تسریع نمود. مسأله بهبود از خطا نیز در نظر گرفته شود.

تمرین ۱۳- گرامر زیر را به فرم LL(1) تبدیل نمایید و سپس با در نظر گرفتن مسأله بهبـود از خطـا بـرای آن یـک تحلیلگـر کاهینـه بازگشتی ایجاد کنید.

S → Abd | Bd
A → Aa | a
B → Bb | b

تمرین ۱۴- گرامر زیر را به فرم LL(1) تبدیل نموده، یک تحلیلگر کاهینه بازگشتی برای آن ایجاد کنید.

S → AB | ABC
A → Aa | l
B → bA | Abb
C → Cc | l

تمرین ۱۵- در حالت کلی چگونه می‌توان اثبات کرد که یک گرامر به فرم LL(1) قابل تبدیل است.

فصل پنجم

تجزیه پایین به بالا

۵-۱- مقدمه

در روش تجزیه پایین به بالا مراحل تجزیه از ترمهای پایانی درون جمله داده شده آغاز، و به سر ترم گرامر خاتمه می یابد .تجزیه گر های پایین به بالا گرامرهای گسترده تری نسبت به تجزیه گر های بالا به پایین را می پو شانند. در این فصل انواع روشهای اتوماتیک تجزیه پایین به بالا مطرح خواهد شد.

چگونگی ترسیم جداول تجزیه، به روشهایی موسوم به، LALR ، SLR ، LR مطرح می گردد. نکته قابل توجه استفاده از گرامر های مبهم است که موضوع آخر این فصل را به خود اختصاص می دهد.

روش LR(K) برای تجزیه پایین به بالا K ترم بعدی را در جمله مورد کامپایل در هر مرحله از تجزیه استفاده می نماید. چون عمل خواندن لغات از داخل جمله ورودی مورد کامپایل از چپ به راست انجام می شود حرف L که مخفف Left to Right می باشد، در نام این روش ظاهر شده است. حرف R مخفف Rightmost derivation یا استنتاج راست است. کلمه LALR مخفف Lookahead LR است. روش LALR(k) نیز برای دسته ای محدودتر از گرامرها نسبت به روش LR(k)مورد استفاده قرار می گیرد. کلمه SLR مخفف Simple LR است. در این فصل روشهای LR(1)، LALR(1)، SLR(1) مورد بررسی قرار خواهد گرفت.

۵-۲- اصول تجزیه پایین به بالا

اصولا" در روش تجزیه پایین بالا پس از اینکه تحلیلگر نحوی لغات را دریافت نمود دو عمل را ممکن است انجام دهد. تحلیلگر یا لغت را مستقیما" به داخل یک پشته به نام پشته تجزیه انتقال می دهد و یا اینکه بر اساس لغت دریافت شده , ترمهای بالای پشته که با سمت راست یک قاعده از قواعد گرامرمطابقت دارند را از بالای پشته خارج نموده ، با ترم میانی سمت راست قاعده جایگزین می کند. با جایگزینی ترمهای بالای پشته با ترم میانی سمت چپ قاعده مربوطه در واقع چند ترم بالای پشته به یک ترم کاهش یافته اند. لذا ،این عمل را در اصطلاح عمل کاهش یا Reduce گویند. عمل انتقال لغات دریافتی از تحلیلگر لغوی به داخل پشته تجزیه را در اصطلاح عمل انتقال یا Shift گویند.

عمل کاهش یا Reduce بر اساس یک قاعده از گرامر انجام می شود. برای نمونه ترمهای بالای پشته تجزیه که عینا مشابه ترمهای سمت راست قاعده شماره مثلا" n هستند، به ترم سمت چپ قاعده که یک ترم میانی یا سرترم گرامر است کاهش داده می شود. این عمل را بطور خلاصه با دستور Rn مشخص می کنند. دستورالعمل Rn مشخص می کند که عمل Reduce بر اساس قاعده شماره n انجام می شود.

اصولا" در روش تجزیه پایین به بالا عمل خواندن لغات مثل قبل از چپ به راست جمله داده شده انجام می شود. ترمها بر اساس قواعد زبان ،دسته بندی وبه یک ترم میانی در سمت راست قواعد کاهش می یابند و یا در اصطلاح خارجی Reduce داده می شوند. یک جمله یا برنامه در داخل یک فایل متن یا در اصطلاح Text قرار می گیرد، در انتهای هر فایل متن علامت خاتمه فایل ظاهر می شود، هر جمله داده شده نهایتا" در صورت صحت از لحاظ فرم گرامری به سرترم گرامر کاهش داده می شود .بنابراین پس از سرترم گرامر همواره انتظار مشاهده علامت خاتمه فایل می رود.

علامت خاتمه فایل در اینجا با کرکتر $ مشخص می شود. جهت کسب اطمینان از ظهور علامت خاتمه فایل پس از سرترم گرامر چنانچه برای نمونه سرترم گرامر S باشد، قاعده

S′→S$

را به ابتدای گرامر افزوده ، در اصطلاح گرامر را به فرم توسعه یافته یا Extended تبدیل می نمایند. بنابر این به ابتدای گرامر عبارت چهار عمل اصلی با سرترم E ،قاعده:

E '→E$

افزوده شده ،در اصطلاح گرامر عبارات به فرم توسعه یافته تبدیل می شود.

0 E '→E$
1 E→E+T
2 E→E-T
3 E→T
4 T→T * F
5 T→T / F
6 T→F
7 F→id
8 F→no
9 F→(E)

بنابر گرامر فوق ،صحت عبارت (a-b) *c/d سنجیده می شود. برای این منظور از یک پشته تجزیه، جهت حفظ فرمهای جمله ای استفاده می شود .

عملیات	ورودي	پشته تجزیه
S,S	(a-b)*c/d $	(a
R7,R6,R3	-b)*c/d $	(E
S,S	-b)*c/d $	(E-b
R7	)*c/d $	(E-F
R6	)*c/d $	(E-T
R2	)*c/d $	(E
S	)*c/d $	(E)
R9	)*c/d $	F
R6	)*c/d $	T
S,S	/d $	T*c
R7	/d $	T*F
R4	/d $	T
S,S	$	T/d
R7	$	T/F
R5	$	T
R3,S,R0	$	E

مراحل تجزیه جمله (a –b)* c / d

در جدول فوق ، نکته قابل توجه انتخاب عمل کاهش و یا انتقال در هر مرحله از تجزیه پایین به بالا است. تصمیم گیری در مورد انتخاب عمل کاهش (Reduce) و یا انجام عمل انتقال (Shift)، به موقعیت کنونی پشته و فرم جمله موجود در آن و بالاخره چگونگی ترم موجود بر سر ورودی که در اصطلاح ترم پیش بینی یا Look ahead نامیده می شود ،وابسته است. برای نمونه در شروع عمل تجزیه ابتدا ´(´ و سپس ´a´ از ورودی خوانده می شود. چون ترم بعدی موجود بر سر ورودی اکنون علامت منها می باشد وطبق گرامر و بنا بر قاعده :

2 E→E-T

حتما قبل از منها یک E باید وجود داشته باشد ، تصمیم به کاهش a بر اساس قاعده شماره هفت ویا بطور مختصر تصمیم به انجام

دستورالعمل R7 و سپس R6 و نهایتا" R3 گرفته شد. لذا ،تا این مرحله درخت تجزیه بصورت زیر ایجاد می شود
در مرحله بعدی با انجام عمل S یا در واقع عمل انتقال لغت ´-´ از سر ورودی خوانده می شود. اکنون طبق گرامر پس از علامت ´-´ باید در ورودی T ظاهر شود . لذا، ترم بعدی یعنی ´b´ از ورودی به داخل پشته تجزیه انتقال یا در اصطلاح Shift د اده می شود.
حالا ، بر سر ورودی ترم ´(´ قرار گرفته است. قبل از ´(´ بر طبق قاعده :

9 F→(E)

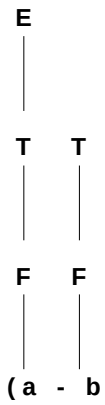
باید E ظاهر شود. تا این مرحله طبق جدول تجزیه بر سر پشته فرم جمله ای

´(E-b´

قرار دارد وترم پیش بینی ´(´ است. لذا ، جهت رسیدن به E قبل از ´(´ عمل R7 و سپس عمل R6 انجام می شود. بنابراین فرم جمله ای موجود در پشته تجزیه به صورت

´(E-T´

تبدیل می شود . تا این مرحله از تجزیه پایین به بالا درخت تجزیه بصورت زیر است. فرمهای جمله ای ایجاد شده نیز در داخل پشته تجزیه در بالا مشخص می باشند.



حالا ترم پیش بینی ´(´ و بر سر پشته تجزیه ترم Tموجود است.اما بلا فاصله T با E جایگزین نشده، دستورالعمل R3 اجرا نمی شود. باید توجه نمود که همواره تنها ترم پیش بینی عامل تصمیم گیرنده نیست بلکه، چگونگی فرم جمله ای در بالای پشته تجزیه نیز در تصمیم گیری و انتخاب نوع عمل کاهش ویا انتخاب عمل انتقال موثر است. لذا، در این مرحله عمل R2 انتخاب شده و سر پشته تجزیه E-T به E بر طبق قاعده شماره۲ کاهش داده می شود. تا این مرحله از تجزیه پایین به بالا درخت تجزیه به صورت زیر است :

۵-۳- طرح روشی برای ایجاد جدول تجزیه LR(1)

در این بخش چگونگی ایجاد الگوریتم تجزیه پایین به بالا استدلال می شود. الگوریتم حاصل بطور خلاصه در انتها ارائه شده است.

الگوریتم حاصل تجزیه LR(1) نامیده می شود. این الگوریتم مبتنی بر تحلیل روش تجزیه که در بالا توضیح داده شد ایجاد میشود.

۵-۳-۱- تولید الگوریتم

در بخش قبل مشاهده کردید که در حالت کلی برای انجام عمل تجزیه نیاز به استفاده از یک پشته تجزیه می باشد. دو عمل صورت می گیرد. یکی انتقال و دیگری کاهش. عمل انتقال و کاهش بر اساس چگونگی ترم پیش بینی بر سر ورودی و وضعیت کنونی پشته تجزیه مشخص می شد. در بعضی از تجزیه گرهای LR نیاز به بیش از یک ترم پیش‌بینی برای عمل تجزیه است، برای نمونه اگر حداکثر K ترم پیش بینی نیاز باشد، تجزیه‌گر را LR(K) می‌نامند.

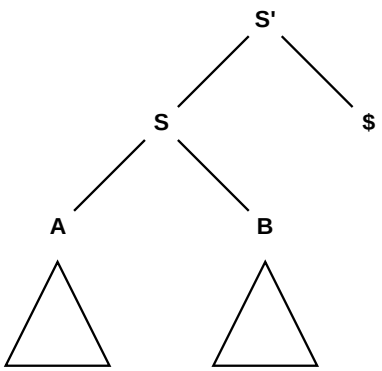
اکنون با یک استدلال ساده نشان داده خواهد شد که چگونه می توان بصورت اتوماتیک بر اساس جدول تجزیه که در مورد آن بحث خواهد شد عمل تجزیه پایین به بالا را انجام داد. برای نمونه گرامر زیر را در نظر بگیرید:

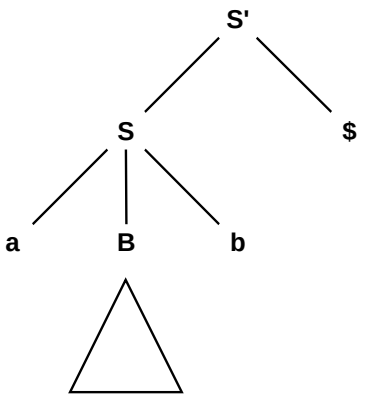
0) S' ⟶ S$
1) S ⟶ aBb
2) S ⟶ AB
3) A ⟶ bA
4) A ⟶ b
5) B ⟶ Bb
6) B ⟶ a

بر طبق گرامر فوق نهایتاً در بالای درخت تجزیه باید سر ترم 'S ظاهر شود. بنابر این آخرین مرحله در تجزیه پایین به بالا کاهش S$ به 'S است. اما قبل از اینکه بتوان این عمل را انجام داد در ورودی ابتدا باید ترم S ظاهر شود. بنابر این می توان گفت در تجزیه پایین به بالا، در آغاز کار تحلیلگر نحوی در انتظار کاهش ورودی یا به عبارت دیگر برنامه یا جمله مورد کامپایل به S است.می توان این وضعیت انتظار را بصورت زیر برای تحلیلگر مشخص نمود

S' ⟶ .S$
1) S ⟶ aBb
2) S ⟶ AB

در این وضعیت که در انتظار S باید بود. اما برای تشکیل S باید طبق گرامر یا عمل R1 یا عمل R2 تجزیه شود. یعنی طبق گرامر رشته aBb یا رشته AB در ورودی ظاهر شود. به عبارت دیگر قبل از خاتمه عمل تجزیه باید یکی از درختهای زیر ایجاد شوند.





پس از مشاهده سر ترم S طبق وضعیت انتظار می **S' ⟶ .S$** رود که علامت خاتمه فایل $ در ورودی ظاهر شود. پس از مشاهده $ است که می توان عمل R0 را انجام داد. بنابر این تا این مرحله می توان گفت که وضعیت در شروع عمل تجزیه بصورت زیر می تواند باشد.

S' ⟶ .S$
S ⟶ .aBb ,$
S ⟶ .AB ,$

در بالا علامت $ توسط ویرکول از دو گسترش متفاوت S جدا شده است . به این ترتیب مشخص شده که پس از کاهش سمت راست این دو قاعده به S ، انتظار می رود که در ورودی $ ظاهر شود. به عبارت دیگر انتظار می رود که پس از تشخیص S ترم پیش بینی $ باشد. این حالت را با وضعیت روبرو مشخص نمود . **S' ⟶ .S$**

بنابر این تا کنون مشخص شده است که در حالت شروع و قبل از آن هیچ لغتی از جمله مورد کامپایل در ورودی ظاهرنشده است، تحلیلگر نحوی انتظار S' را دارد، لذا جهت تشکیل S' در آغاز انتظار کاهش ورودی به S را دارد. پس از مشاهده سر ترم S انتظار می رود که علامت خاتمه فایل ($) در ورودی ظاهر شود .

I0:	**S' ⟶ .S$**
	S ⟶ .aBb ,$
	S ⟶ .AB ,$
	A ⟶ .bA ,First(B)
	A ⟶ .b ,First(B)

برای مشاهده S در آغاز کار انتظار مشاهده a و یا تشکیل ترم میانی A می رود . طبق وضعیت :

S ⟶ .aBb ,$

انتظار می رود ابتدا a و سپس B، و پس از آن b در ورودی ظاهر شوند تا بتوان عمل R1 را انجام داد و S را ایجاد کرد. پیش بینی R1 در صورتی مؤثر است که در سر ورودی $ وجود داشته باشد. به همین ترتیب طبق وضعیت باید ابتدا A ایجاد شود پس از آن B، تا بتوان عمل R2 را انجام داد **S ⟶ .AB ,$** . بنابر این باید ابتدا در ورودی A تشخیص داده شود و پس از تشخیص A باید حتماً در سر ورودی ترم متعلق به مجموعه First(B) ظاهر شود تا بتوان مطمئن شد که می توان B را پس از A دید .

برای ایجاد A نیز طبق گرامر باید با عمل R3 و یا R4 انجام شود . و بعد از انجام عمل حالا بر طبق انتظاری که در بالا طبق وضعیت می رفت، تحلیلگر نحوی باید یک ترم متعلق به **S ⟶ .AB ,$** First(B) را در ورودی مشاهده کند تا بتواند به کار خود ادامه دهد (B را در ورودی تشخیص دهد). بنابر این جهت رسیدن به A یکی از دو وضعیت زیر در آغاز کار و قبل از دریافت هر گونه لغتی از تحلیلگر لغوی پیش بینی می شود.

بنا بر این در حالت شروع و قبل از دریافت اولین لغت از تحلیلگر لغوی وضعیت های مورد انتظار تحلیلگر نحوی بصورت زیر می باشد.

A ⟶ .bA ,First(B)
A ⟶ .b ,First(B)

حال فرض کنید که اولین ترم پایانی یا اولین لغت از تحلیلگر لغوی دریافت شود. چه اتفاقی می تواند رخ دهد؟ طبـق انتظـار تحلیلگـر (که در بالا مشخص شده است) این لغت باید b یا a باشد. طبق وضعیت :

S ⟶ .aBb ,$

انتظار مشاهده a و طبق وضعیتهای:

A ⟶ .bA ,First(B)
A ⟶ .b ,First(B)

در آغاز انتظار مشاهده b می رود . اگر در ورودی b ظاهر شود حالت جدید زیر به وجود می آید.

I1: A ⟶ b.A ,First(B)
A ⟶ b. ,First(B)

حالا اگر در ورودی First(A) ظاهر شود باید به سراغ گسترش A در ادامه وضعیت b•A رفت، و اگر First(B) در ورودی ظاهر شود باید طبق قاعده $A \rightarrow \bullet\, b$ عمل کاهش b به A را انجام داد.

بنابر وضعیت برای مشـاهده A **A ⟶ b.A ,First(B)** در این حالـت، انتظـار دیـدن A دومـی هـم در سـمت راسـت وضعیت می رود. اما برای مشاهده A در ورودی طبق قواعد ۳ و ۴ گرامر باید bA یا b در ورودی ظاهر شوند. بعد از این حالت باید در ورودی First(B) ظاهر شود. بنابر این حالت I1 به صورت زیر تبدیل می شود.

I1: A ⟶ b.A ,First(B)
A ⟶ b. ,First(B)
A ⟶ b.A ,First(B)
A ⟶ b. ,First(B)

A ⟶ b.A ,First(B)

A ⟶ b.A ,First(B)
A ⟶ b. ,First(B)

در حالت I1 طبق وضعیت انتظار مشاهده A پس از b در سمت راست وضعیت می رود،همچنین پس از A انتظار مشـاهده First(B) می رود. بنابر این، برای تشکیل A در ورودی دو وضعیت :

به حالت 11 افزوده شده است. به این ترتیب در این حالت طبق یکـی از دو وضـعیت فـوق A در ورودی تشـخیص داده خواهـد شـد (وضعیتی که در انتظار تشکیل A در ورودی است) و از این حالت تغییر حالت داده و حالت جدید I5 تشکیل می شود .

حال می توان عمل R3 را انجام داد.

I5: A ⟶ bA. ,First(B)

در حالت I1 طبق وضعیت ، مشخص شده که **A ⟶ b.A ,First(B)** در ورودی b ظـاهر شـده اسـت. اکنـون، در صورتیکه ترم پیش بینی عنصری متعلق به First(B) باشد، می توان ادعا کرد که در ورودی A مشاهده شده است. بنابر این می تـوان به حالت I0 برگشت و اعلام کرد که طبق انتظار A در ورودی مشاهده شد.

اکنون ، در حالت I0 وضعیت ،که در انتظار مشاهده **S ⟶ A.B ,$** A در ورودی بوده به وضعیت جدید:

S ⟶ A.B ,$

تبدیل می شود . این وضعیت هسته حالت جدید تری بنام I2 بصورت زیر می شود .

I2: S ⟶ A.B ,$	
B ⟶ .Bb ,$,b	
B ⟶ .a ,$,b	

در هسته حالـت I2 طبـق وضـعیت انتظـار **S ⟶ A.B ,$** مشاهده B می رود. پس از B نیز همانند S انتظار مشاهده $. بنابر این باید گسترشهای مختلف b با ترم پیش بینی $ را ایجاد کرد. پس دو وضعیت زیر به این حالت افزوده می شود :

B ⟶ .Bb ,$
B ⟶ .a ,$

B ⟶ .Bb ,$

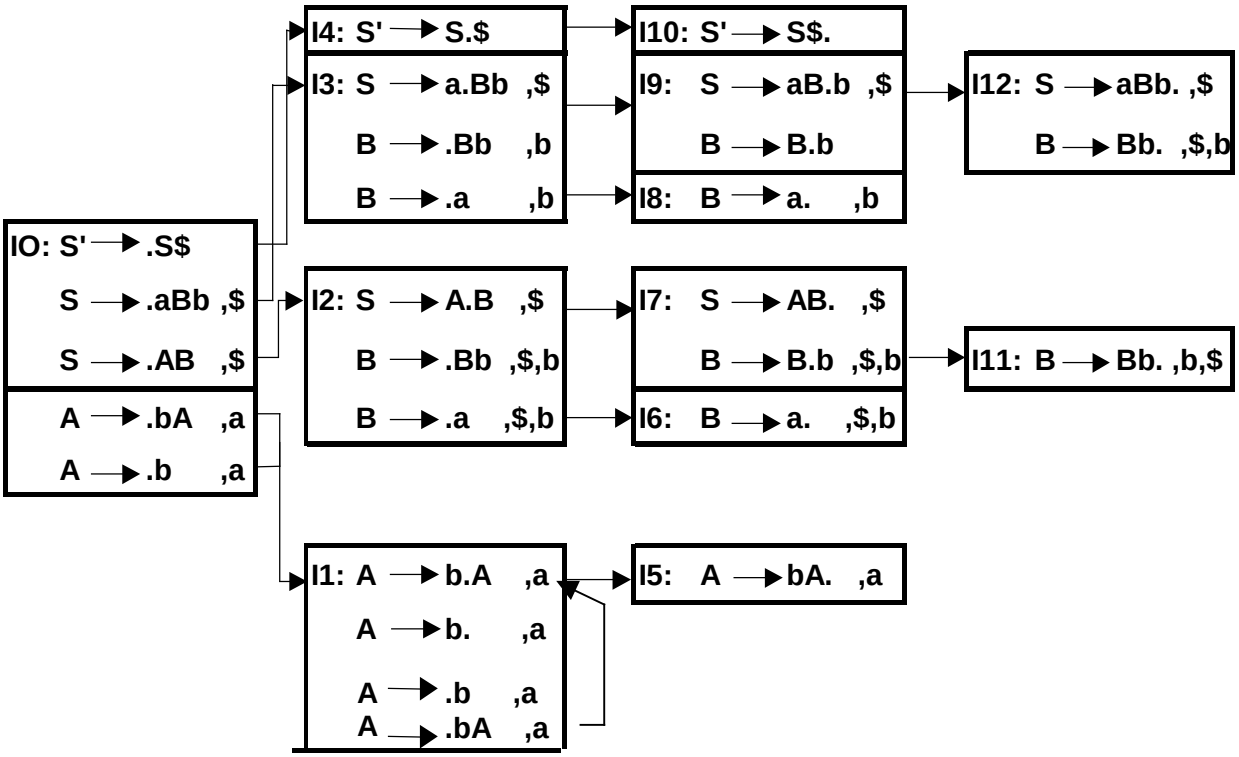
حال با توجه به وضعیت انتظار مشاهده B در سمت راست ترم می رود، و پس از B انتظار دیدن b در ورودی. برای مشـاهده B طبـق گرامر باید Bb یا a در ورودی ظاهر شوند. بنابر این باید دو وضعیت زیر به این حالت افزوده شوند.

B ⟶ .Bb ,b
B ⟶ .a ,b

اما چون از قبل این دو وضعیت با ترم پیش بینی $ وجود دارند، و می توان ترم پیش بینی b را فقط به آن افزود. بنابر این دو وضعیت زیر به حالت I2 افزوده شده است.

B ⟶ .Bb ,$,b
B ⟶ .a ,$,b

اگر به این ترتیب ادامه داده شود ، گراف تجزیه LR(1) بصورت زیر ایجاد می شود:



شکل ۵-۱: گراف تجزیه LR(1)

۵-۳-۲- الگوریتم تولید گراف تجزیه LR(1)

بطور خلاصه برای تولید گراف حالات که یک نمونه از آن در شکل بالا نمایش داده شد، باید مراحل زیر طی شوند:

۱- عمل تولید گراف از ایجاد حالت شروع I0 آغاز می شود. چنانچه سر ترم گرامر S باشد، در هسته یا در اصطلاح Kernel این حالت وضعیت

I0: S' → .S$.

می شود ، علامت • قبل از S مشخص می کند که در آغاز انتظار مشاهده S می رود .

۲-برای هر وضعیت

$A \longrightarrow \alpha.B\beta \ ,\delta$

از یک حالت که در آن a و یا هر رشته ای از ترم ها و یا اینکه تهی می تواند باشد، باید کلیه گسترشهای ترم میانی B بصورت زیر وضعیتهای:

$B \longrightarrow \bullet\eta \quad , FIRST(\beta)$

را به آن حالت افزود. ترم پیش بینی برای این گسترشها (FIRST(β است و چنانچه β وجود نداشته باشد ، آنگاه ترم پیش بینی δ است. برای نمونه یکی از وضعیت های حالت I0 در گراف شکل فوق وضعیت **S → .AB ,$** است. چون پس از • ترم

میانی A قرار دارد، گسترشهای مختلف A با ترم پیش بینی First(B) که مساوی a است، در نظر گرفته شده است.

A ⟶ .bA ,a
A ⟶ .b ,a

۳ـ ترم پیش بینی برای وضعیت های خود بازگشتی با فرم کلی :

$A \longrightarrow \bullet A\,\alpha \quad ,\delta$

$A \longrightarrow \bullet \beta \quad ,\delta$

بصورت:

$A \longrightarrow \bullet A\,\alpha \quad ,\delta, \text{First}(\alpha)$

$A \longrightarrow \bullet \beta \quad ,\delta, \text{First}(\alpha)$

است . برای نمونه در حالت I2 از گراف شکل ۱،۵ بـرای وضـعیت طبـق معمـول دو **S ⟶ A.B ,$**

گسترش ترم میانی B با ترم پیش بینی S ایجاد می شود:

B ⟶ .Bb ,$
B ⟶ .a ,$

اکنون پس از • ترم میانی B وجود دارد لذا، باید وضعیتهای:

B ⟶ .Bb ,b
B ⟶ .a ,b

را به حالت 12 افزود ، بنابر این با ترکیب این دو وضعیت با وضعیت های بالا دو وضعیت زیر حاصل می شود :

B ⟶ .Bb ,$,b
B ⟶ .a ,$,b

۵-۳-۳- خلاصه عملیات در گراف تجزیه

اگر به گراف شکل ۵-۱ توجه نمایید. سه نوع عملیات در آن مستتر است.

۱ـ عمل کاهش یا Reduce : عمل کاهش در بالا توضیح داده شد.برای نمونه در حالت I11 طبق وضـعیت چنانچـه بـر سـر ورودی یکی از دو ترم پایانی$ یا b ظاهر شود، می توان **B ⟶ Bb. ,b,$** بر طبق قاعده عمل کاهش را انجام داد یا بـه عبـارت دیگـر می توان طبق قاعده شماره ۵ عمل R5 را انجـام داد. در حالـت کلـی **B ⟶ Bb** عمل Rn به مفهوم کاهش بر اساس قاعـده شماره n است .

۲- عمل انتقال shift: عمل دیگری که در گراف مستتر است، عمل انتقال است. برای نمونه در حالت I0 با مشاهده b در ورودی، b به داخل پشته تجزیه انتقال داده شده، و سپس به وضعیت I1 تغییر حالت داده می شود. این عمل را با نماد S1 نمایش می دهند. در حالت کلی عمل Sn به مفهوم انتقال و سپس گذر به حالت n است.

۳- عمل GoTO: عمل سوم عمل رفتن مستقیم یا در اصطلاح Goto از یک حالت به یک حالت دیگر است. برای نمونه در حالت I0 با تشکیل A در سر پشته بلافاصله گذری به حالت ۲ انجام می شود (عمل Goto2 انجام می شود). در حالت کلی با تشکیل یک ترم میانی مورد انتظار بر روی پشته می توان Go to را به حالت جدیدتر انجام داد.

۵-۳-۴- ایجاد جدول تجزیه

جدول تجزیه در واقع نمایش ماتریس گراف تجزیه است. عملیات انتقال و Goto موجب تغییر حالت و یا گذر بین گراف می شود. برای نمونه جدول تجزیه گراف شکل ۵-۱ را می توان به جدول زیر تبدیل نمود.

	a	b	$	S	A	B
0	S3	S1	-	4	2	
1	-	S1	-	-	5	-
2	S6	-	-	-	-	7
3	S8	-	-	-	-	9
4	-	-	S10	-	-	
5	R3	-	-	-	-	-
6	-	R6	R6	-	-	-
7	-	S11	R2	-	-	-
8	-	R6	-	-	-	-
9	-	S12	-	-	-	-
10	-		ACCEPT		-	-
11	-	R5	R5	-	-	-
12	-	R5	R1	-	-	-

جدول تجزیه LR(1)

اکنون با استفاده از جدول فوق می توان به راحتی عمل تجزیه بالا به پایین را انجام داد، برای نمونه عبارت aabbb را توسط جدول فوق بصورت زیر می توان مورد تجزیه پایین به بالا قرار داد.

پشتهٔ تجزیه	ورودي	دستورالعمل
0	aabbb$	S3
0a3	abbb$	S8
0a3a8	bbb$	R6
0a3B	bbb$	Goto 9
0a3B9	bbb$	S12
0a3Bb12	bb$	R5
0a3B	bb$	Goto 9
0a3B9	bb$	S12
0a3Bb12	b$	R5
0a3B	b$	Goto 9
0a3B9	b$	S12
0a3B9b12	$	R1
0s	$	Goto 4
0s4	$	S10
0s4$10		پذیرش
S		

مراحل تجزیه LR(1)

مثال : برای مشخص شدن چگونگی عمل تجزیه LR(1) به مثال زیر توجه نمایید:

S ⟶ CC
C ⟶ aC
C ⟶ d

گرامر ساده فوق را در نظر بگیرید، هدف ایجاد جدول تجزیه LR(1) برای این گرامر است. همانگونه که قبلاً گفته شد، ابتدا باید گرامر را به فرم توسع یافته تبدیل نمود و پس از آن قواعد را شماره گذاری کرد. به این ترتیب گرامر فوق به صورت زیر تبدیل می شود:

اکنون می توان در مورد حالات ممکن تجزیه شروع به استدلال نمود. به این ترتیب که در

0) S' ⟶ S$
1) S ⟶ CC
2) C ⟶ ac
3) C ⟶ d

S' ⟶ .S$

حالت شروع , ما در وضعیت قرار داریم. در اینجا علامت '.' پس از فلش، چگونگی وضعیت را مشخص می کند. هدف تشـخیص S ودر نهایت مشاهده علامت $ می باشد.

پس باید در حالت شروع، در ورودی یک S را مشاهده کرد (انتظار دیدن S می رود)، و پس از آن انتظار دیدن علامت $ می رود. اما طبق گرامر برای مشاهدهS باید در ورودیCC دیده شود، لذا وضعیت در حالت شروع به حالت زیر تبدیل می شود:

S' ⟶ .S$
S ⟶ .CC

طبق وضعیت ، پس از S انتظار دیدن $ می رود. پس، پیش بینی می شود که بعد از C نیز در ورودی علامت $ مشاهده شود. حال در این وضعیت انتظار می رود که در اولین مرحله در ورودی یکC و بعد از این C یک C دومی نیز مشاهده شود:

$$S' \longrightarrow .S\$$$ $$S \longrightarrow .CC\ ,\$$

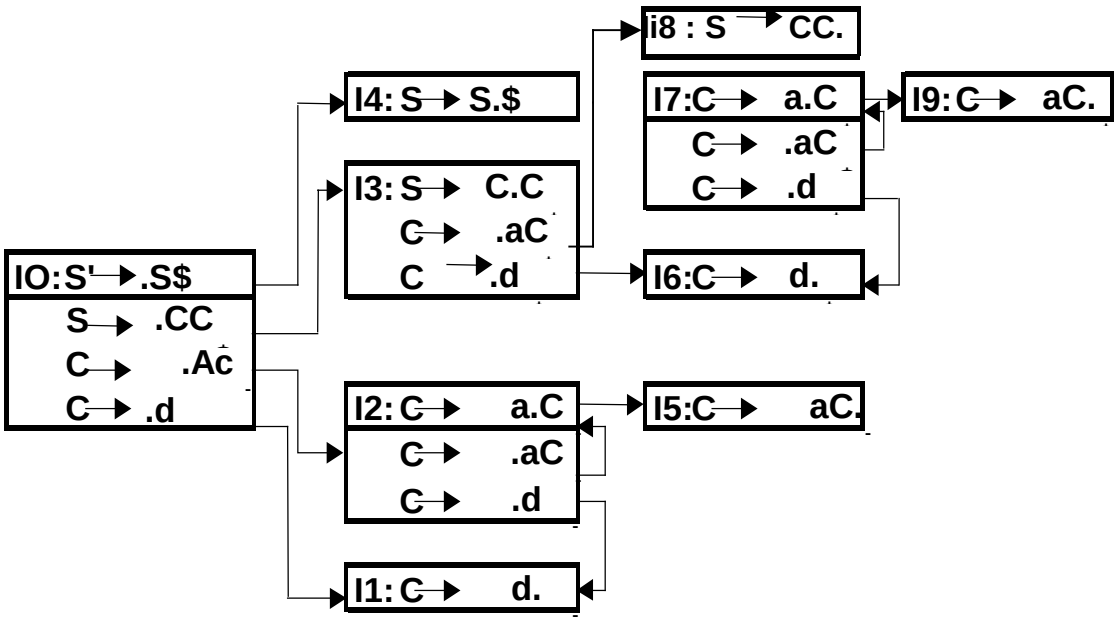
$$S' \longrightarrow .S\$$$
$$S \longrightarrow .CC\ ,\$$$
$$C \longrightarrow .aC\ ,First(C)$$
$$C \longrightarrow .d\ ,First(C)$$

همانگونه که مشاهده می شود، طبق وضعیت انتظار مشاهده یک C در ورودی را داریم. لذا، باید C در ورودی تشخیص داده شود. برای

$$C \longrightarrow .d\ ,First(C)$$

تشخیص C باید طبق گرامر یا aC و یا ترم پایانی d مشاهده شوند. در حالت شروع اگر در ورودی d ظاهر شود حالت جدید I1 تولید می شود. حالت I1 تنها شامل وضعیت:

$S \longrightarrow .CC$ می باشد. لذا، پس از مشاهدهd در ورودی می توان، به حالت I0 بازگشت و اعلام کرد که در وضعیت اانتظار مشاهده یکC می رفت و حالا در ورودیC تشخیص داده شده اسست. به این ترتیب، حالت جدید I3 با وضعیت در هسته أن ظاهر می شود. پس اینجا تلاش برای دیدنC دوم در ورودی و مشاهده $ بعد از C دوم أغاز می شود. بطور کلی گراف تجزیه بر طبق استدلال فوق بصورت زیر ایجاد می شود:



گراف تجزیه LR(1)

که جدول تجزیه برای گراف فوق بصورت زیر می باشد:

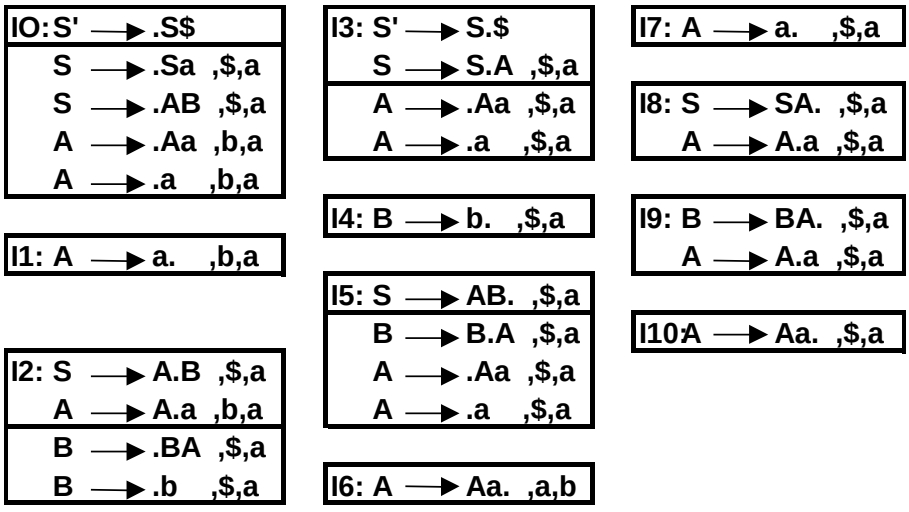
	Action			Go To	
	a	d	$	S	C
O	S2	S1	-	4	3
1	R3	R3	-	-	-
2	S2	S1	-	-	5
3	S7	S6	-	-	8
4	Accept			-	-
5	R2	R2	-	-	-
6	-	-	R3	-	-
7	S7	S6	-	-	9
8	-	-	R1	-	-
9	-	-	R2	-	-

شکل ۵-۵ جدول تجزیه LR(1)

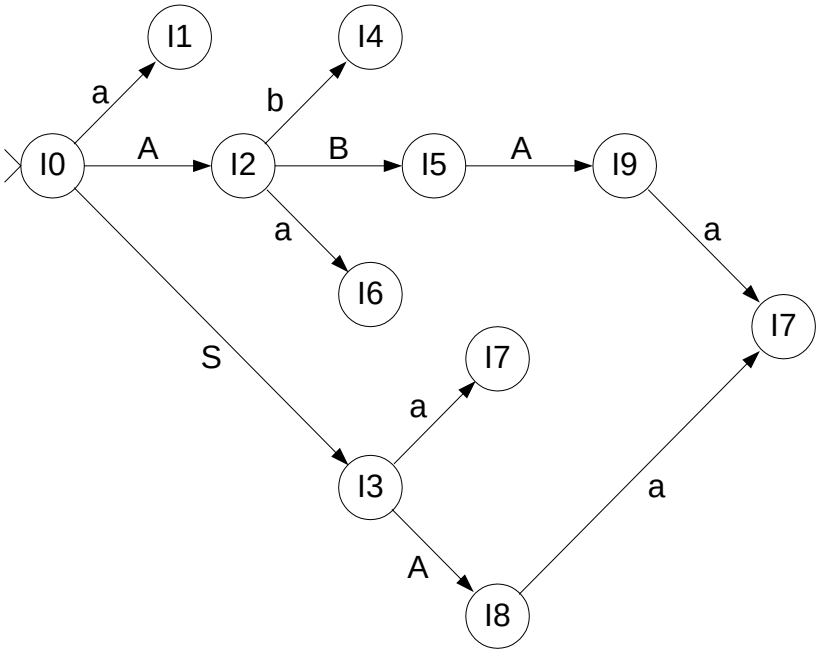
مثال : جدول تجزیه برای LR(1) گرامر زیر ایجاد کنید:

S' ⟶ S$
S ⟶ SA
S ⟶ AB
B ⟶ BA
B ⟶ b
A ⟶ Aa
A ⟶ a

همانگونه که مشاهده می شود گرامر فوق در فرم توسعه یافته می باشد، پس می توان گراف تجزیه را بلافاصله ایجاد کرد. چون تعداد حالات ممکن نسبتا زیاد می باشد، هر یک از حالات را بصورت جداگانه مشخص نموده سپس با استفاده از یک گراف ارتباط بین حالات را ترسیم می کنیم.



گراف تجزیه بصورت زیر است:



جدول تجزیه LR(1) و گرامر در زیر ارایه شده است.

	a	b	$	S	A	B
O	S1	-	-	13	12	-
1	R6	R6	-	-	-	-
2	S6	S4	-	-	-	5
3	S7	-	پذیرش	-	8	-
4	R4	-	R4	-	-	-
5	R2 , S7	-	R2	-	9	-
6	R5	R5	-	-	-	-
7	R6	-	R6	-	-	-
8	R1,S10	-	R1	-	-	-
9	R3,S10	-	R3	-	-	-
10	R5	-	R5	-	-	-

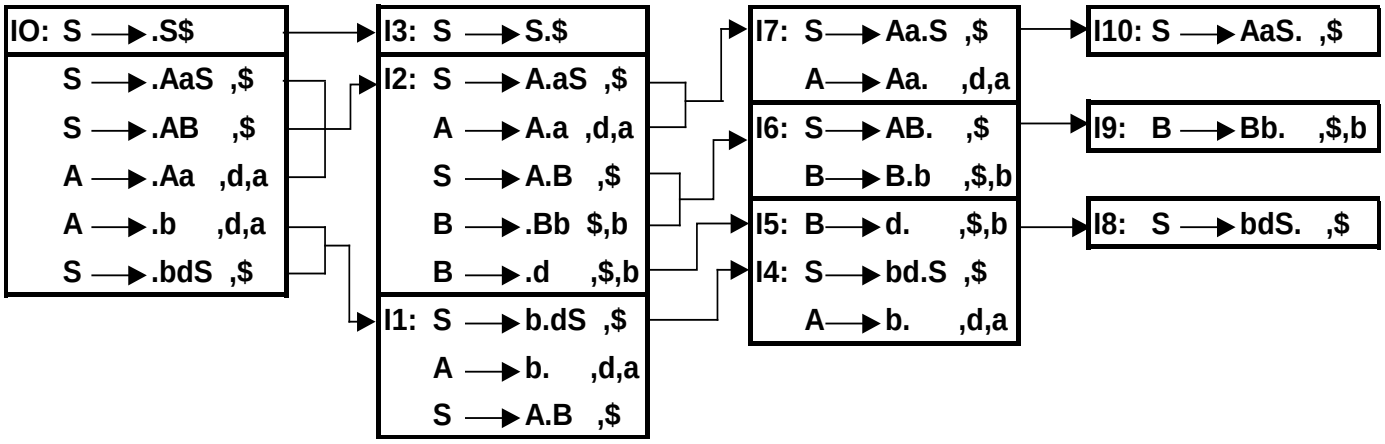
همانگونه که مشاهده می شود، در سه حالت I5, I8 و I9 در داخل جدول تجزیه مشخص نیست که با مشاهده a آیا باید عمل کاهش و یا عمل انتقال را انجام داد. در اصطلاح، در این سه حالت اختلال در کاهش و انتقال وجود دارد، لذا، گرامر LR(1) نیست.

اختلال در کاهش و انتقال را در اصطلاح Shift Reduce Conflict نیز می گویند. این مشکل موجب می شود که با در دست داشتن یک ترم پیش بینی نتوان تصمیم گرفت عمل Shift باید انجام شود یا عمل Reduce، ممکن است Reduce Reduce Conflict نیز در برخی موارد ایجاد شود.

مثال : جدول تجزیه LR(1) برای گرامر زیر ایجاد کنید:

S' ⟶ S$
S ⟶ AaS
S ⟶ bdS
S ⟶ AB
A ⟶ Aa
A ⟶ b
B ⟶ Bb
B ⟶ d

گراف تجزیه LR(1) برای گرامر ذکر شده بصورت زیر است:



گرامر فوق LR(1) نیست. زیرا در حالت I1 با مشاهده d در ورودی مشخص نیست که آیا عمل R5 و یا عمل S4 باید انجام شود. بعبارت دیگر نمی توان مشخص کرد که آیا عمل انتقال باید انجام شود و یا عمل کاهش، بنابراین،در این حالت اختلال در کاهش و انتقال و یا در اصطلاح Shift Reduce Conflict وجود دارد.

۵-۴- مشکل گرامرهای LR(1)

اگر توجه نموده باشید در گرامرهای LR(1) جدول تجزیه نسبت به تعداد قواعد آن بسیار بزرگ می باشد. برای نمونه برای گرامر کوچک:

S ⟶ CC
C ⟶ aC
C ⟶ d

جدول تجزیه دارای ۱۰ ردیف و ۵ ستون بود. به این ترتیب جدول تجزیه برای یک گرامر با فقط ۳ قاعده تعداد ۵۰ خانه داشت. برای گرامرهای واقعی با حدود ۱۰۰ قاعده، جدول تجزیه حداقل ۱۰۰۰۰ خانه از حافظه را اشغال می کند. جهت رفع این مشکل سعی شده

E ⟶ E + T | E - T | T
T ⟶ T * F | T / F | F
F ⟶ id | no | '(' E')'

که گرامر را محدود کنند و گرامرها با امکانات کمتر از گرامرهای LR(1) جهت بیان قواعد استفاده شود و یـا اینکـه گرامرهـا بصـورت مبهم استفاده شوند. برای مثال گرامر عبارات بصورت:

را می توان بصورت مبهم و با یک ترم میانی بصورت زیر تعریف کرد:

همانطور که مشاهده می کنید، در این گرامر بجای ۳ ترم میانی E, T, F فقط یک ترم میانی استفاده شده، و بجای ۹ قاعده برای فرم کلی عبارات ۷ قاعده استفاده شده است، که جدول را بسیار کوچکتر می نماید، البته ابهام در گرامر مشکلات قابل حلی بوجـود خواهـد

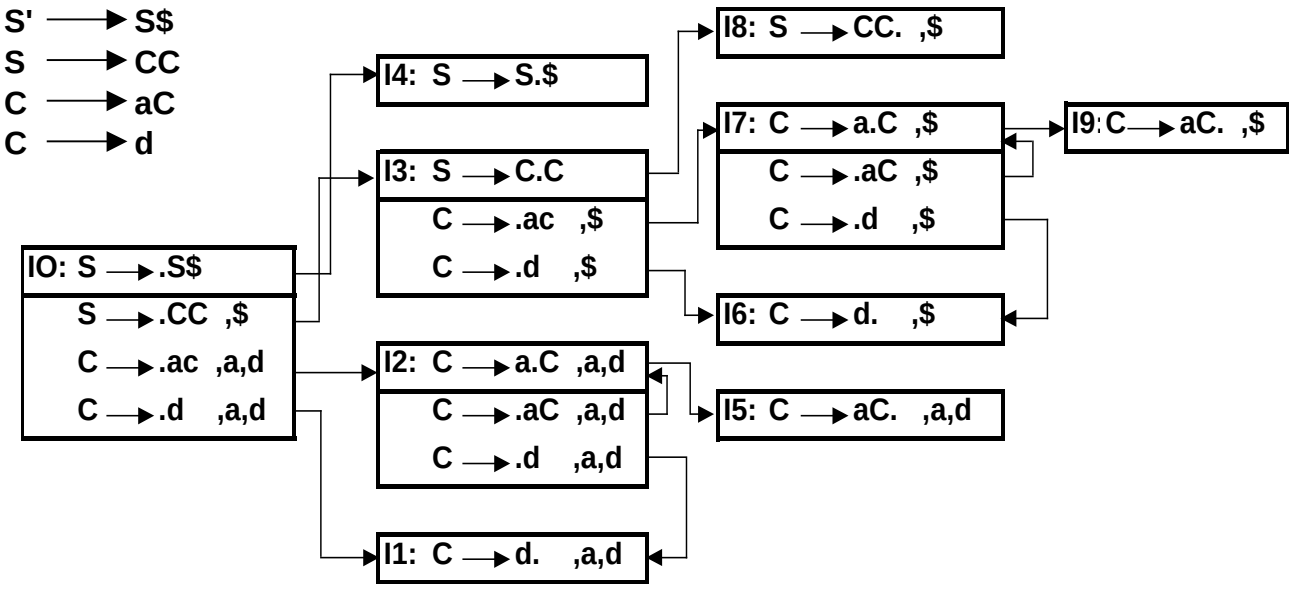
E ⟶ E + E | E - E | E * E | E / E | '(' E ')' | id | no

أورد که در مورد آنها کاملا بحث خواهد شد.

روش دیگر برای کوچکتر نمودن جدول تجزیه پایین به بالا، استفاده از گرامرهای محدودتری به نام LALR وSLR می باشد. این نـوع گرامرها در ادامه بحث توضیح داده خواهد شد.

۵-۵- گرامرهای LALR(1)

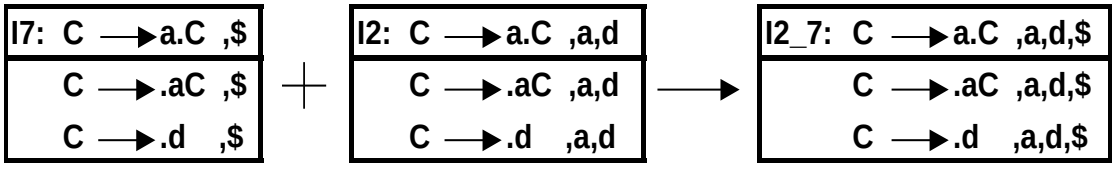
در هنگام تولید جدول تجزیه LR(1) چنانچه بتوان بدون هیچ گونه مشکلی حالات با وضعیتهای مشابه، اما ترمهای پیش بینی متفاوت را با یکدیگر ادغام نمود. گرامر را LALR(1) می نامند. مشکل به این صورت می تواند باشد که با در دست داشتن یک ترم پیش بینـی یا بعبارت دیگر یک ورودی بعدی نتوان تصمیم گرفت که چه عملی باید صورت بگیرد. برای نمونه نتوان تصـمیم گرفـت کـه آیـا عمـل کاهش باید انجام شود یا عمل انتقال. برای درک بهتر مسئله به گرامر و گراف تجزیه مربوط به آن در زیر توجه کنید.



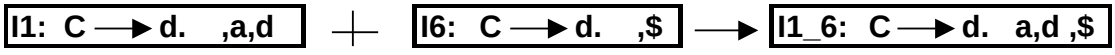
گراف تجزیه LR(1)

در گراف فوق دو حالت I2 و I7 دارای وضعیتهای یکسان با ترمهای پیش بینی متفاوت هستند. حالات I1 و I6 و همچنین حالات I5 و I9 نیز به همین صورت می باشند. اگر این حالات را با یکدیگر ادغام نمایید. حاصل بصورت زیر خواهد بود:

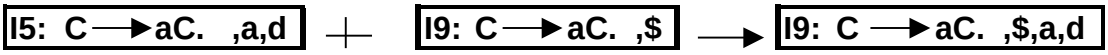
الف- حالت I2 با I7 ادغام شده، حالت I2-7 ایجاد می شود.



ب- حالت I1 با حالت I6 ادغام شده، حالت I1-6 ایجاد می شود.



ج- حالت I5 با حالت I9 ادغام شده حالت جدید I5-9 ایجاد می شود.



اکنون در جدول تجزیه باید به جای حالتهای ادغام شده، حالت جدید ادغامی را قرار دهیم.

برای نمونه در بالا حالتهای I5 و I9 از جدول تجزیه LR(1) حذف و با I5-9 جایگزین می شود. به این ترتیب، هر دستورالعمل انتقال S5 یا S9 با دستورالعمل S5-9 جایگزین می شود. هر عمل Goto به I5-9 تبدیل می شود. جدول تجزیه قبل از جایگزینی و پس از جایگزینی حالتهای ترکیب شونده، در زیر مشخص شده است:

الف - قبل از ادغام

	Action			Go To	
	a	d	$	S	C
O	S2	S1	-	4	3
1	R3	R3	-	-	-
2	S2	S1	-	-	5
3	S7	S6	-	-	8
4	Accept			-	-
5	R2	R2	-	-	-
6	-	-	R3	-	-
7	S7	S6	-	-	9
8	-	-	R1	-	-
9	-	-	R2	-	-

ب- پس از ادغام

	Action			Go To	
	a	d	$	S	C
O	S2_7	S1_6	-	4	3
1	R3	R3	R3	-	-
2	S2_7	S1_6	-	-	5_9
3	S2_7	-	S1_6	-	8
4	Accept			-	-
5	R2	R2	R2	-	-
6	-	-	R1	-	-

قبل از ادغام باید حالتهای کاندید را در داخل جدول مشخص کرد. هر جا که دستورالعمل Goto به یکی از حالتهای ادغامی وجود دارد، نام حالت ادغامی به جای آن قرار می گیرد. همین عمل برای دستورالعملهای انتقال یا در اصطلاح Shift نیز باید انجام داد.

مثال : جدول تجزیه LALR(1) برای گرامر زیر ایجاد کنید:

E' ⟶ E$
E ⟶ E + T
E ⟶ T
T ⟶ T * F
T ⟶ F
F ⟶ (' E ')
F ⟶ id

حالات گراف تجزیه LR(1) برای این گرامر در زیر مشخص شده است:

IO: E' ⟶ .E$	**I7:** F ⟶ (.E) ,*,+,)	**I13:** F ⟶ (E.) ,*,+,)
E ⟶ .E+T ,+,$	E ⟶ .E+T ,+,)	E ⟶ E.+T ,+,)
E ⟶ .T ,+,$	E ⟶ .T ,+,)	**I14:** T ⟶ T*.F , *,+,)
T ⟶ .T*F ,*,+,$	T ⟶ .T*F ,*,+,)	F ⟶ .(E) ,*,+,)
T ⟶ .F ,*,+,$	T ⟶ .F ,*,+,)	F ⟶ .id ,*,+,)
F ⟶ .(E) ,*,+,$	F ⟶ .(E) ,*,+,)	**I15:** E ⟶ E+.T ,),+
F ⟶ .id ,*,+,$	F ⟶ .id ,*,+,)	T ⟶ .T*F ,*,+,)
I1: F ⟶ id. ,$,+,*	**I8:** T ⟶ F. ,),*,+	T ⟶ .F ,*,+,)
I2: F ⟶ (.E) ,$,+,*	**I9:** E ⟶ T. ,+,)	F ⟶ .(E) ,*,+,)
E ⟶ .E+T ,),+	T ⟶ T.*F ,*,+,)	F ⟶ .id ,*,+,)
E ⟶ .T ,),+	**I10:** F ⟶ (E.) ,*,+,$	**I16;** F ⟶ (E). ,*,+,$
T ⟶ .T*F ,*,+,)	E ⟶ E.+T ,+,)	**I17:** T ⟶ T*F. ,*,+,$
T ⟶ .F ,*,+,)		**I18:** E ⟶ E+T. ,+,$
F ⟶ .(E) ,*,+,)	**I11:** T ⟶ T*.F ,*,+,$	T ⟶ T.*F ,*,+,$
F ⟶ .id ,*,+,)	F ⟶ .(E) ,*,+,$	**I19:** F ⟶ (E). ,*,+,)
I3: T ⟶ F. ,*,+,$	F ⟶ .id ,*,+,$	**I20:** T ⟶ T*F. ,+,*,)
I4: E ⟶ T. ,$,+	**I12:** E ⟶ E+.T ,+,$	**I21:** E ⟶ E+T. ,*,+,)
T ⟶ T.*F ,*,+,$	T ⟶ .T*F ,*,+,$	T ⟶ T.*F ,*,+,)
I5: E' ⟶ E.$	T ⟶ .F ,*,+,$	
E ⟶ E.+T ,$,+	F ⟶ .(E) ,*,+,$	
I6: F ⟶ id. ,),+,*	F ⟶ .id ,*,+,$	

در زیرجدول تجزیه LR(1) و LALR(1) برای این گرامر ایجاد شده است:

	Action						Go To		
	id	(	)	+	*	$	F	T	E
O	S1	S2	-	-	-	-	3	4	5
1	-	-	-	R6	R6	R6	-	-	-
2	S6	S7	-	-	-	-	8	9	10
3	-	-	-	R4	R4	R4	-	-	-
4	-	-	-	R2	S11	R2	-	-	-
5	-	-	-	S12	-	Accept	-	-	-
6	-	-	R6	R6	R6	-	-	-	-
7	S6	S7	-	-	-	-	8	9	13
8	-	-	R4	R4	R4	-	-	-	-
9	-	-	R2	R2	S14	-	-	-	-
10	-	-	S16	S15	-	-	-	-	-
11	S1	S2	-	-	-	-	17	-	-
12	S1	S2	-	-	-	-	3	18	-
13	-	-	S19	S15	-	-	-	-	-
14	S6	S7	-	-	-	-	20	-	-
15	S6	S7	-	-	-	-	8	21	-
16	-	-	-	R5	R5	R5	-	-	-
17	-	-	-	R3	R3	R3	-	-	-
18	-	-	-	R1	S11	R1	-	-	-
19	-	-	R5	R5	R5	-	-	-	-
20	-	-	R3	R3	R3	-	-	-	-
21	-	-	R1	R1	S14	-	-	-	-

الف- جدول تجزیه LR(1)

	Action						Go To		
	id	(	)	+	*	$	F	T	E
O	S1	S2	-	-	-	-	3	4	5
1	-	-	R6	R6	R6	R6	-	-	-
2	S1	S2	-	-	-	-	3	4	10
3	-	-	R4	R4	R4	R4	-	-	-
4	-	-	R2	R2	S11	R2	-	-	-
5	-	-	-	S12	-	Accept	-	-	-
10	-	-	S16	S12	-	-	-	-	-
11	S1	S2	-	-	-	-	17	-	-
12	S1	S2	-	-	-	-	3	18	-
16	-	-	R5	R5	R5	R5	-	-	-
17	-	-	R3	R3	R3	R3	-	-	-
18	-	-	R1	R1	S11	R1	-	-	-

ب- جدول تجزیه LALR(1) حاصل از ترکیب حالات زیر:

(1, 6) - (2, 7) - (3, 8) - (4, 9) - (10, 13) -(14, 11) - (15, 12) - (16, 19) - (17, 20) - (18, 21)

برای ادغام حالات، ابتدا باید حالاتی را در نظر گرفت که بدون ایجاد مسئله اختلال در کاهش و انتقال، بسادگی قابل ادغام هستند. پس، حالات I1 وI6 و I3 و I8 و I16 و I19 و I17 و I20 بلافاصله قابل ادغام هستند، با جایگزینی حالات معادل مشاهده می شود که حالات I11 و I14 نیز قابل ادغام هستند، با ادغام این دو حالت می توان نتیجه گرفت که دو حالت I18 و I21 نیز قابل ادغام هستند.

۵-۶- گرامرهای SLR(1)

گرامرهای Simple LR(1) یا بطور خلاصه SLR(1) روش دیگری برای تجزیه پایین به بالا می باشد،

این نوع گرامرها بسیار محدودتر از گرامرهای نوع LALR(1) و در نتیجه گرامرهای LR(1) می باشند. زیرا، در این نوع از گرامرها فرض بر این است که ترمهای پیش بینی برای یک ترم میانی A وابسته به محل قرار گرفتن A در سمت راست قواعد نمی باشد، بلکه برای هر ترم میانی A، مستقل از چگونگی قرار گیری آن در سمت راست قواعد مختلف، همواره مجموعه Follow(A) شاخص ترمهای پیش بینی برای A است. بنابراین تعداد حالات یا ردیفهای جدول تجزیه برابر با جدول LALR(1) است. زیرا در اینجا همانند LALR(1) ترمهای پیش بینی عاملی برای تفکیک دو حالت از یکدیگر نیستند، یا بعبارت دیگر نمی توان دو حالت مشخص نمود که دارای وضعیتهای یکسان اما ترمهای پیش بینی متفاوت باشند.

بعنوان مثال، برای گرامر زیر جدول تجزیه SLR(1) ایجاد می کنیم:

0) E' ⟶ E$
1) E ⟶ E + T
2) E ⟶ T
3) T ⟶ T * F
4) T ⟶ F
5) F ⟶ (' E ')'
6) F ⟶ id

در مورد گرامرهای SLR(1) قبل از ایجاد گراف تجزیه باید اقدام به تولید مجموعه پیرو یا در اصطلاح Follow برای ترمهای میانی نمود تا برای هر ترم، ترمهای پیش بینی آن مشخص شود.

مجموعه پیرو برای ترمهای گرامر فوق بصورت زیر می باشد:

Follow(E) = { $, +,) }
Follow(T) = { * } + Follow(E) = { $, +,), * }
Follow(F) = Follow(T)

اکنون با در دست داشته مجموعه پیرو برای ترمهای میانی بسادگی می توان گراف تجزیه SLR(1) را تولید نمود.

I0:	E' → .E$	→ I5	
	E → .E+T	→ I5	
	E → .T	→ I4	
	T → .T*F	→ I4	
	T → .F	→ I3	
	F → .(E)	→ I2	
	F → .id	→ I1	
I1:	F → id.		
I2:	F → (.E)	→ I6	
	E → .E+T	→ I6	
	E → .T	→ I4	
	T → .T*F	→ I4	
	T → .F	→ I3	
	F → .(E)	→ I2	
	F → .id	→ I1	
I3:	T → F.		
I4:	E → T.		
	T → T.*F	→ I7	
I5:	E' → E.$		
	E → E.+T	→ I8	
I6:	F → (E.)	→ I9	
	E → E.+T	→ I8	
I7:	T → T *.F	→ I10	
	F → .(E)	→ I2	
	F → .id	→ I1	
I8:	E → E+.T	→ I11	
	T → .T*F	→ I11	
	T → .F	→ I3	
	F → .(E)	→ I2	
	F → .id	→ I1	
I9:	F → (E).		
I10:	T → T*F.		
I11:	E → E+T.		
	T → T.*F	→ I7	

در زیر جدول تجزیه SLR(1) برای گرامر فوق ارایه شده است. باید توجه داشته باشید که برای این دسته از گرامرها مجموعه های پیرو ترمهای پیش بینی را ایجاد می کنیم، لذا در جدول تجزیه برای کاهش id به F بر طبق قاعده شماره ۶ در زیر ستونهای مربوط به عناصر مجموعه پیرو F دستورالعمل R6 قرار داده شده است.

	Action						Go To		
	id	(	)	+	*	$	F	T	E
O	S1	S2	-	-	-	-	3	4	5
1	-	-	R6	R6	R6	R6	-	-	-
2	S1	S2	-	-	-	-	3	4	6
3	-	-	R4	R4	R4	R4	-	-	-
4	-	-	R2	R2	S7	R2	-	-	-
5	-	-	-	S8	-	Accept	-	-	-
6	-	-	S9	S8	-	-	-	-	-
7	S1	S2	-	-	-	-	10	-	-
8	S1	S2	-	-	-	-	3	11	-
9	-	-	R5	R5	R5	R5	-	-	-
10	-	-	R3	R3	R3	R3	-	-	-
11	-	-	R1	R1	S7	R1	-	-	-

جدول تجزیه SLR(1) برای گرامر ساده عبارات

۵-۷- گرامرهای مبهم

همانگونه که قبلاًتوضیح داده شد، گرامر مبهم گرامری ست که بر اساس آن بیش از یک درخت تجزیه برای جمله داده شده، و در نتیجه دو مفهوم متناقض برای آن بتوان داشت. با استفاده از گرامرهای مبهم می توان اندازه جدول تجزیه پایین به بالا را کوچکتر

نمود. برای نمونه گرامر ساده عبارات را که در بخشهای قبل جدول تجزیه LR(1) و LALR(1) و SLR(1) برای آن ایجاد شد را بخاطر بیاورید. جدول تجزیه SLR(1) و LALR(1) برای این گرامر دارای یازده حالت و در مجموع ۱۰۸ خانه از حافظه را اشغال می کرد، با نوشتن گرامر عبارات بصورت مبهم زیر:

E ⟶ E + E | E - E | E * E | E / E | (E) | id | no

تعداد ترمهای میانی کاهش می یابد، در ضمن نشان داده خواهد شد که مبهم بودن گرامر مشکلاتی را نیز بدنبال خواهد داشت:

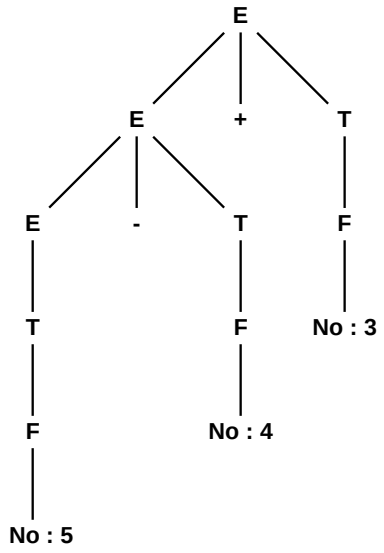
الف-در گرامرهای مبهم اولویت تمام عملگرها یکسان است(اولویت عملگر "ضرب یا تقسیم"، با "جمع و منها" یکسان است).

ب- اجتماع عملگرهای "جمع و ضرب" مشخص نیست.

قواعد مربوط به گرامر عبارات در حالت غیر مبهم بیانگر اولویت و اجتماع عملگرهاست.

در گرامر فوق "جمع و منها" دارای اجتماع چپ می باشند. برای نمونه درخت تجزیه برای عبارت 3 + 4 - 5 را در نظر بگیرید:

0) E' ⟶ E$
1) E ⟶ E + T
2) E ⟶ E - T
3) E ⟶ T
4) T ⟶ T * F
5) T ⟶ T / F
6) T ⟶ F
7) F ⟶ (' E ')'
8) F ⟶ id
9) F ⟶ no



همانگونه که مشاهده می کنید ابتدا باید عبارت 4 - 5 را محاسبه نمود تا بتوان پس از آن حاصل را با 3جمع کرد. در واقع باید از چپ به راست عمل جمع و تفریق انجام شود. این بخاطر بیان قواعد مربوط به E بصورت خود بازگشتی چپ است. اگر قواعد مربوط به E بصورت خود بازگشتی راست بیان می شد، یعنی بصورت:

E ⟶ T + E | T - E | T

آنگاه عبارت 3 + 4 - 5 از راست به چپ محاسبه می شد، ابتدا مقدار۴ + ۳ و سپس (3 + 4) - 5 محاسبه می شد. بر طبق گرامر فوق قبل از اینکه بتوان E را محاسبه کرد، باید T محاسبه شود، زیرا درون T عملیات "ضرب و تقسیم" و در درون E عملیات "جمع

و منها" تعریف شده اند. لذا طبق گرامر فوق عملیات "ضرب و تقسیم" بر عملیات "جمع و منها" اولویت دارند. با بیـان گرامـر خلاصـه عبارات بصورت مبهم:

0) E' ⟶ E$
1) E ⟶ E + E
2) E ⟶ E * E
3) E ⟶ (E)
4) E ⟶ id

اولاً، اولویتها یکسان می شوند، ثانیاً، اجتماع عملگرها نامشخص خواهد بود. نامشخص بودن اجتماع عملگرها و یکسان بودن اولویت آنها درهنگام ایجاد جدول تجزیه SLR(1) برای گرامر مبهم عبارات مشکلاتی را ایجاد می‌کند.

IO:	E' ⟶ .E$	I4:	F ⟶ (E.)	I7:	E ⟶ (E).
	E ⟶ .E+E		E ⟶ E.+E	I8:	E ⟶ E*E.
	E ⟶ .E*E		E ⟶ E.*E		F ⟶ E.+E
	E ⟶ .(E)	I5:	E ⟶ E*.E		E ⟶ E.*E
	E ⟶ .id		E ⟶ .E+E	I9:	E ⟶ E+E.
I1:	E ⟶ id.		E ⟶ .E*E		F ⟶ E.+E
I2:	E ⟶ (.E)		E ⟶ .(E)		E ⟶ E.*E
	E ⟶ .E+E		E ⟶ .id		
	E ⟶ .E*E	I6:	E ⟶ E+.E		
	E ⟶ .(E)		E ⟶ .E+E		
	E ⟶ .id		E ⟶ .E*E		
I3	E' ⟶ E.$		E ⟶ .(E)		
	E ⟶ E+.E		E ⟶ .id		
	T ⟶ E*.E				

جدول تجزیه برای گراف فوق که در شکل زیر آمده دارای دو نوع عملکرد برای عملگرهای "جمع و ضرب" در حالات 9 و 8 می باشـد، در این دو حالت که اختلال در کاهش و انتقال وجود دارد و مشخص نیست که با مشاهده دو عملگر "جمع و ضرب" آیا عمل کاهش را باید انجام داد یا انتقال را. باید توجه داشت که در گرامرهای SLR(1) ترمهای پیش بینی را مجموعه های پیرو مشخص می کننـد. در گرامر فوق برای ترم میانی E باید مجموعه پیرو برای آن محاسبه شود.

	Action						Go To
	id	(	)	+	*	$	E
O	S1	S2	-	-	-	-	3
1	-	-	R4	R4	R4	R4	-
2	S1	S2	-	-	-	-	5
3	-	-	-	S6	S5	Accept	-
4	-	-	S7	S6	S5	-	-
5	S1	S2	-	-	-	-	8
6	S1	S2	-	-	-	-	9
7	-	-	R3	R3	R3	R3	-
8	-	-	R2	S6,R2	S5,R2	R2	-
9	-	-	R1	S6,R1	S5,R1	R1	-

جدول تجزیه SLR(1) برای گرامر مبهم عبارات

در حالت 8 با توجه به وضعیت **E ⟶ E * E.** میتوان گفت که قبلاً یک علامت * در ورودی دیده شده است. طبق وضعیت **E ⟶ E .+ E** انتظار مشاهده + در سر ورودی می رود. لذا، با مشاهده یک عملگر + در ورودی چون عملگر * بر + اولویت دارد عمل کاهش R2 انجام می شود. برای نمونه عبارت 4 + 3 * 2 را در نظر بگیرید. مسلماً در اینجا قبل از عمل انتقال علامت + به داخل پشته تجزیه، باید عمل ضرب انجام شود، لذا، به جای S6 عملR2 انجام می شود.

در حالت 9 با توجه به وضعیت **E ⟶ E + E.** می توان گفت که قبلاً یک علامت + در ورودی دیده شده است. طبق وضعیت E→E.*Eانتظار مشاهده * در سر ورودی می رود، لذا، با مشاهده یک عملگر * در وردوی چون عملگر "ضرب" بر "جمع" اولویت دارد، عمل انتقال S5 انجام می شود. برای نمونه عبارت 4 * 3 + 2 را در نظر بگیرید. مسلماً در اینجا قبل از انجام عمل "جمع" باید عمل "ضرب" انجام شود، لذا، عملگر* به داخل پشته تجزیه انتقال داده شده، بجای R1 عمل S5 انجام می شود.

در حالت 8 با توجه به وضعیت **E ⟶ E * E.** می توان گفت که قبلاً یک علامت * در ورودی دیده شده است. طبق وضعیت **E ⟶ E .* E** انتظار مشاهده * دیگری در سر وروردی می رود، لذا، با مشاهده یک عملگر * در ورودی چون"ضرب"دارای اجتماع چپ است، عمل کاهش R2 انجام می شود.

برای نمونه عبار ت 4 * 3 * 2 را در نظر بگیرید. مسلماً در اینجا قبل از انتقال علامت * به داخل پشته تجزیه، باید عمل عمل "ضرب" انجام شود، لذا بجای S5 عمل R2 انجام می شود.

در حالت 9 با توجه به وضعیت **E ⟶ E + E.** می توان گفت که قبلاً یک علامت + در ورودی دیده شده است. طبق وضعیت انتظار مشاهده + **E ⟶ E .+ E** دیگری در سر ورودی می رود، لذا با مشاهده یک عملگر + در ورودی چون جمع دارای اجتماع چپ است، عمل کاهش R1 انجام می شود. برای نمونه عبارت 4 + 3 + 2 در نظر بگیرید. مسلماً در اینجا قبل ار انتقال علامت + به داخل پشته تجزیه، باید عمل جمع انجام شود، لذا بجای S6 عمل R1 انجام می شود.

مثال: برای گرامر مبهم جملات شرطیIf جدول تجزیه SLR(1) ایجاد کنید.

0) S' ⟶ S$
1) S ⟶ I S e S
2) S ⟶ I S
3) S ⟶ a

در گرامر فوق S بیانگر جمله، I مخفف If و e مخفف else است.

در اولین مرحله، باید مجموعه پیرو را برای ترمهای میانی و سر ترم گرامر بدست بیاوریم، که این مجموعه در واقع ترمهای پیش بینی را برای ترم میانی مربوطه مشخص می کند.

Follow(S) = { $, e }

مجموعه حالات بصورت زیر می باشد:

I0:	**S' ⟶ .S$**
	S ⟶ .I S e S
	S ⟶ .I S
	S ⟶ .a
I1:	**S ⟶ a.**
I2:	**S ⟶ I.SeS**
	S ⟶ I.S
	S ⟶ .ISeS
	S ⟶ .IS
	S ⟶ .a

I3:	**S' ⟶ S.$**
I4:	**S ⟶ IS.eS**
	S ⟶ IS.
I5:	**S ⟶ ISe.S**
	S ⟶ .ISeS
	S ⟶ .IS
	S ⟶ .a
I6:	**S ⟶ ISeS.**

	I	e	a	$	S
O	S2	-	S1	-	3
1		R3	-	R3	-
2	S2	-	S1	-	4
3	-	-	-	Accept	-
4	-	S5,R2	-	R2	-
5	S2	-	S1	-	6
6	-	R1	-	R1	-

جدول تجزیه SLR(1) برای گرامر مبهم جملات شرطیIf

در ردیف ۴ از جدول تجزیه چون Else یا e متعلق به نزدیکترین If است، S5 انتخاب شده است.

۵-۸- پرسش و پاسخ

۱- روش LR(K) چیست به چه صورت عمل میکند ؟

۲- اصول کار در تجزیه پایین به بالا چیست ؟

۳- الگوریتم تولید گراف تجزیه LR(1) برای مثال ۱-۵ را توضیح دهید ؟

۴- Shift Reduce Conflict را توضیح دهید ؟

۵- مشکل گرامرهای LR(1) چیست ؟

۶- به چه نوع گرامرهایی , LALR(1) گفته میشود؟

۷- به چه نوع گرامرهایی , SLR(1) گفته میشود؟

۸- به چه نوع گرامرهایی مبهم گفته میشود؟

۵-۹- تمرین

تمرین ۱- نشان دهید که گرامر زیر LALR(1) است؟

S ⟶ Aa | bac | dc | bda
A ⟶ d

تمرین ۲- أیا گرامر زیر از نوع LALR(1) می باشد؟

S ⟶ SAB | aB
A ⟶ aA | Ad
B ⟶ Ba | bAd

تمرین ۳- أیا گرامر زیر LALR(1) است؟

S ⟶ aAB | SDb
A ⟶ aDB | Ab
B ⟶ Bda | abD
D ⟶ Da | db

تمرین ۴- چرا جدول تجزیه LALR(1) و SLR(1) برای یک گرامر به یک اندازه هستند؟

تمرین ۵- چرا اگر گرامری LR(1) باشد و حالالت مشابه آن با یکدیگر ادغام شوند تا جدول تجزیه LALR(1) حاصل شود، در هنگام ادغام حالات، تنها امکان مشکل اختلال در کاهش و انتقال، و نه اختلال در کاهش و کاهش (Reduce Reduce Conflict) به وجود خواهد أمد.؟

تمرین ۶- نشان دهید که هر گرامر LL(1) یک گرامر LR(1) است.

تمرین ۷- با در نظر گرفتن اینکه عملگر "توان" اولویت بیشتری نسبت به "ضرب و تقسیم" دارد و اجتماع آن راست می باشد، گرامر عبارات را تکمیل نموده، جدول تجزیه SLR(1) برای این گرامر ایجاد کنید.

تمرین ۸- در گرامر زیر علامت "^" نمایانگر عملگر "توان" می باشد، جدول تجزیه SLR(1) برای أن ایجاد کنید.

تمرین ۹- جدول تجزیه SLR(1) برای گرامر زیر ایجاد کنید.

E ⟶ E + E | E - E | E * E | E / E | E ^ E | (E) | no | id

$E \longrightarrow E\ \theta 1\ E \mid E\ \theta 2\ E \mid \ldots \mid E\ \theta n\ E \mid (\ E\ ) \mid id$

فرض کنید که عملگرهای θi دارای اجتماع راست می باشند و اگر اندیس i بزرگتر از اندیس j باشد، θi دارای اولویت بیشتری نسبت به θj دارد.

تمرین ۱۰- نشان دهید که گرامر زیر LL(1) است و SLR (1) نیست.

S ⟶ AaAb | BbBa
A ⟶ λ
B ⟶ λ

آیا گرامر فوق LR(1) است؟

راهنمایی - توجه داشته باشید که در هر حالتی که گسترش تهی برای ترمهای میانی A یا B وجود داشته باشد. می توان در داخل جدول تجزیه در زیر ستون مربوط به ترمهای پیش بینی برای این دو ترم عمل کاهش قرار داد.

فصل ششم

تحلیل مفهومی

تحلیل مفهومی در حد Type Checking انجام می گیرد. زبانهایی که تحلیل مفهومی را انجام می دهند، اصطلاحا Strongly Typed Languages نامیده می شوند. در این زبانها معمولا هر متغیر از نوع خاص تعریف می شود. زبانهایی وجود دارد که در آنها متغیرها معرفی نمی شوند، اما عمل Type Checking انجام می شود، در این دسته از زبانها نوع هر متغیر وابسته به نوع تخصیص داده شده با آن می باشد. برای مثال، اگر در داخل متغیر A مقداری از integer نوع باشد، نوع متغیر A نیز integer می باشد، ولی اگر در زمان اجرا یک نوع Real به آن اختصاص بدهیم نوع متغیر A به Real تغییر می کند، لذا به این دسته از زبانها Dynamic Typed Languages می گویند.

در زبانها عادی عمل آزمون نو در زمان کامپایل برای این منظور با استفاده از یک کلاس به نام Symbol Tables هر گاه که متغیری در داخل جدول نمادها گنجانده می شود و در هنگامی که در داخل عبارت یا جملات یک متغیر استفاده می شود، نوع آن از داخل جدول نمادها مورد بررسی قرار می گیرد.

همانگونه که در بالا گفته شد، تحلیل مفهومی در حد آزمون نو صورت می گیرد، برای این منظور می‌بایست جدول نمادها را ایجاد کنیم.

در ادامه به ذکر چگونگی ایجاد آن و پر کردن آن در ضمن عمل تحلیل نحوی می پردازیم.

```
Const C1 = 5;
Type
  Student = record
    a : array [1..10] of integer;
    b : real;
  End;
  F = record
    i : integer;
    j :array [1..10] of char;
  End;
Var
  a : integer;
  d : real;
  c : char;
  T : Student;
```

در ضمن عمل کامپایل با استفاده از کلاس Symbol Table در هنگام تحلیل نحوی با مشاهده تعریف (Decleration) اسامی تعریف شده در داخل جدول نمادها گنجانده می شوند. اما ابتدا ابزار کار یعنی کلاس Symbol Table باید مشخص شود.

۱- تعیین ساختار کلاس (Data Structure) :

با توجه به مثال ارایه شده مشخص می کتیم که چه فیلدهایی برای تعیین دقیق ساختار جدول نمادها ضروری است.

(فرض می کنیم طول هر کلمه ۲ بایت است و فضای ذخیره سازی ۲ کیلو بایت)

Name	Token	Group	Offset/Value	Type
C1	-	S_Const	[5]	int
Student	-	S_Type	-	.
a	-	S_Var	0	integer
b	-	S_Var	2	real
C1	-	S_Var	6	char
f	-	S_Var	8	.
T	-	S_Var	00??	.

Record Descriptor		
Name	Offset	Type
a	0	.
b	20	real

Array Descriptor	
Element _Type	: int
Lower_Bound	: 1
Upper_Bound	: 10
No_Dim	: 1

```
class Buket {
        string Key; object Binding; Buket Next;
    Bucket(string k, object  b, Bucket n) { key = k; binding := b; next := n;}        }


class HashT{
        final int size = 256;
        Bucket tabel[] = new Bucket[size];
         Int hash(string S_ )
         {
           int h = 0;
           for (int I = 0; I < s.length; I++)
           h := h * 65599 + s.charAt( I );  //S_ ام در I کاراکتر
           return h;
         }
         void insert(string S_, binding b)
         {
           int index = hash(S_) % size;
           table[index] = new Bucket(S_, b, table[index]);
         }
         object LookUp(string S_)
         {
           int index = hash(S_) % size;
           for (binding b = table[index]; b != null; b := b.next)
           if (s.equal = S_(b.key)) return b.binding;
           return null;
         }
         void pop(string S_)
         {
           int index = hash(S_) % size;
           table[index] := table[index].next;
```

```
    }
}// end of class HashT
```

در مثال فوق مشاهده می کنید که جدول نمادها بصورت Hash Table و در قالب کلاسی به نام HashT ارایه شده، و هر ردیف آن در قالب کلاسی به نام Bucket معین شده است.

Size بعنوان یک Const مطرح شده است، توابع Insert و LookUp بیشتر از همه مورد نیار تحلیلگر مفهومی می باشند، تحلیلگر مفهومی با استفاده از تابع LookUp به دنبال نام متغیر می گردد و نوع ان را پیدا می کند، تابع Insert کار افزودن نانم جدید را به داخل جدول نمادها را بر عهده دارد.

نکته در اینجاست که چگونه کامپایلر با بهره برداری از توابع Insert و LookUp جدول نمادها را ایجاد می کند و برای تحلیلگر مفهومی مورد استفاده قرار می دهد.

```
Program→Program id ';' BlockBody'.'
BlockBody→[ConstDefpart] [TypeDefpart]
          [VarDefpart] { ProDefpart | FunDefpart}
          compound Stop

(* ConsDefpart → const ConstDef {ConstDef} *)
Procedure ConstDefpart(Stop : Stops);
Begin
//مانند قبلی
End;

(* ConstDef → id '=' ( id | no )';' *)
Procedure ConstDef(Stop : Stops);
Begin
  c := CurrentToken;
  SymTable.Insert(CurrentToken);
  Expect(S_id, Stop + [S_equal]);
  Expect(S_equal, Stop + [S_id, S_no]);
  If CurrentSymbol = S_no then
    Begin
      SymTable.Update(c, CurrentToken.Lexeme, S_var);
      NextSymbol;
    End
  Else
    Begin
      d := LookUp(CurrentToken);
      SymTable.Update(c, d.Value, S_var);
      Expect(S_id, Stop + [S_semicolon]);
    End;
  Expect(S_semicolon, Stop);
End;

(* VarDefpart → Var VarDef {VarDef} *)
Procedure VarDefpart(Stop  : Stops);
Begin
//مانند قبلی
End;

(* VarDef →  id {, id} ':' (integer | real) ';' *)
Procedure VarDef(Stop : Stops);
Var
  s : stack;
```

```
Begin
  s.push(CurrentToken);
  Expect(S_id, …);
  While(CurrentSymbol = S_comma) do
    Begin
      NextSymbol;
      s.push(CurrentToken);
      Expect(S_id, …);
    End;
  Expect(S_colon, Stop + …);
  While not (s.isEmpty) do
    SymTable.Insert(s.pop, CurrentToken.Lexeme);
  If CurrentSymbol = S_real then
    NextSymbol
  Else
    Expect(S_int, Stop);
End;

(* F→ id | no |'('E')' *)
Procedure F(Stop : Stops; Var ResF : string; R : integer; TypeF : Symbols);
Begin
  If CurrentSymbol = S_id then
    Begin
      TypeF := SymTable.LookUp(CurrentToken);
      R := 0;
      ResF := CurrentToken.Lexeme;
    End
  Else If CurrentSymbol = S_no then
    Begin
      TypeF := S_const_int;
      R := 0;
      ResF := CurrentToken.Lexeme;
    End
  Else
    Begin
      Expect(S_OpenPar, …);
      E(Stop + [S_ClosePar], ResF, R, TypeF);
      Expect(S_ClosePar, …);
    End;
End;

(* T→ F { (* | / ) F} *)
Procedure T(Stop : Stops; Var ResT : string; R : integer; TypeT : Symbols);
Var
  OpCode, Type1 : Symbols;
  Operand1 : string;
  R1 : integer;
Begin
  F(Stop + [S_mul, S_div], ResT, R, TypeT);
  While (CurrentSymbol = S_mul) or (CurrentSymbol = S_div) do
  Begin
    OpCode := CurrentSymbol;
    NextSymbol;
    If (R =0) then
    Begin
      R := 1;
      Operand := NewTemp;
```

```
    Emitln('mov ' + Operand + ',' + ResT);
    ResT := Operand;
  End;
  F(Stop + [S_mul, S_div], Operand, R1, Type1);
  If OpCode = S_ mul then
    Emit('mul ')
  Else
    Emit('div ');
  Emitln(ResT + ',' + Operand);
  If R = 1 then
    RemTemp;
  Type1 := Compare(TypeT, Type1);
 End;
End;
```